School of Mathematics and Systems Engineering

Reports from MSI - Rapporter från MSI

Retrieval and Analysis of Software Systems from SCM Repositories

Michael Müller

Sep
2007

MSI
Växjö University
SE-351 95 VÄXJÖ

Report 07111
ISSN 1650-2647
ISRN VXU/MSI/DA/E/--07111/--SE

Abstract

One source of input data for software evolution research is data stored inside a software configuration management repository. The data includes different versions of a software system's source code as well as version history metadata, such as check-in dates or log messages. Inherently, extracting this data manually is a time- and labor intensive task. The subsequent preprocessing step and the appropriate storage of the results, necessary to utilize the data for further analysis, is an additional effort for the researcher.

The goal of this thesis is to design and implement a front-end plug-in for an existing software comprehension tool, the VIZZANALYZER, providing the capability to extract and analyze multiple versions and evolutional information of software systems from SCM repositories and to store the results. Thereby, the implemented solution provides the infrastructure for software evolution research.

Keywords: Kenyon, Software evolution, Software configuration management, VizzAnalyzer

Table of Contents

List of Figures

List of Tables

List of Code

Glossary

API	Application Programming Interface
AST	Abstract Syntax Tree
AWT	Abstract Window Toolkit
CVS	Concurrent Versions System
GUI	Graphical User Interface
JDBC	Java Database Connectivity
JDK	Java Development Kit
ORM	Object-Relational Mapping
SCM	Software Configuration Management
SQL	Structured Query Language
SVN	Subversion
UML	Unified Modeling Language
XML	Extensible Markup Language

1 Introduction

Today it is a well known fact in the field of software engineering that any successful software product becomes old. Parnas describes this as a phenomenon of software products that closely resembles human aging [1].

One obvious cause of this so-called software aging is that old products are not updated to meet changing needs. Unless software is frequently updated, its users will become dissatisfied and call the product old and outdated. Consequently, any successful software product has to be changed in order to meet new demands. This includes adding new functionality, accommodating new hardware and repairing faults. However, the results of these changes are simultaneously the second cause of software aging: After many changes a critical point is reached when each new release decreases the maintainability and increases the complexity of the system.

This form of aging happens because software is often changed by people who do not understand the original design concept. Therefore, the changes are likely to be inconsistent with the original concept and, as a consequence, the structure of the program will degrade [1]. New exceptions are added to the existing design rules and, after many such changes, even the original designers are not able to understand the modified product anymore. Along comes the problem that the documentation is either not updated or updated in an inadequate way. Hence, the documentation becomes increasingly inaccurate, making future changes even more difficult. The consequences of this "decay" are that the software product becomes expensive to update, as changes take more time and are more likely to introduce new bugs. Eventually, new releases are not feasible anymore, as the costs they would require are too high.

The phenomenon of software aging has been investigated in several independent studies. Already in 1976, Belady and Lehman observed and discussed the development of IBM's operating system OS/360 and its subsequent enhancements and releases [2]. In a more recent work, Gall et al. examined the evolution of a large Telecommunication Switching System (TSS) over a total of 20 system releases [3]. Eick et al. used the change history of a different, fifteen-year old TSS for their studies on code decay [4].

Even though Parnas suggests several preventive techniques to delay the decay and limit the effects of software aging [1], the outcome of the studies mentioned above show that additional approaches to encounter the problem of increasing complexity and decreasing quality of software are highly desirable. One approach to reach this goal is to restructure the software in order to preserve its maintainability [3].

A prerequisite for any restructuring effort, however, is the identification of those modules of the affected software that need to be restructured. Such modules can be identified with the help of data stored in Software Configuration Management (SCM) repositories. These repositories are rich sources for distinctive kinds of evolutional analyses, as they reflect the reasons and effects of particular changes made to the software system over a certain period of time.

The goal of this thesis is to design and implement a front-end plug-in for an existing software comprehension tool, providing the capability to extract and analyze multiple versions and evolutional information of software systems from SCM repositories and to store the results. Thereby, the implemented solution provides the infrastructure for software evolution research.

1.1 Problem

The existing software system VIZZANALYZER [5] is a reverse engineering framework to compose code retrieval, analysis and visualization tools [6]. However, it is only capable of retrieving and analyzing the source code of one static version of a software system at

a time, which has to be stored manually on the local file system. Automated retrieval and analyzing of several versions is currently not supported. A static version gives information about quantity and quality of a software system at a certain point in time but it lacks information about the evolution of the system, which provides the basis for software evolution research. The version history of a software system is typically retained in a SCM repository. Not only offer such repositories access to different branches and versions of a system but they also store information about its evolution.

Thus, the problem addressed by this thesis is:

Customize a front-end plug-in for the VIZZANALYZER framework, providing the capability to extract and analyze multiple versions and evolutional information of software systems from SCM repositories and to store the results.

Today, a number of SCM systems exist that are commonly used in both commercial and non-commercial software projects. Hence, the implemented solution should offer support for at least two of the most common SCM systems and should also be extendible in this respect. To simplify this task the use of existing software tools offering access to SCM repositories is encouraged. The problem is concretized in Section 1.3 *Goals and Criteria* below.

1.2 Motivation

The implemented solution of the thesis allows the users of the VIZZANALYZER tool to access SCM repositories. There are two reasons behind this intention: One purpose is to automatically retrieve multiple versions of a software system and to invoke an analysis plug-in of the VIZZANALYZER framework on each of these versions. This automation comprises a huge benefit, as it is both time- and labor-intensive to achieve this task manually. The other reason is to gain access to the evolutional information stored inside SCM repositories. This metadata associated with the archived data includes valuable information about the evolution of a software system.

The results of the retrieval and analysis process in association with the evolutional information extracted from a SCM repository can be used as input to further analyses. Existing analysis tools could be plugged into the VIZZANALYZER framework to exploit these results or new tools could be developed. Thus, the implemented VIZZANALYZER plug-in provides the basis for software evolution research which, as discussed above, is a helpful method to encounter the problem of software aging. Section 6.2 *Future Work* briefly covers some possibilities of the plug-in.

1.3 Goals and Criteria

As mentioned in Section 1.1, the aim of this thesis is to extend an existing software system, the VIZZANALYZER, with the ability to automatically retrieve and analyze source code and evolutional information from SCM repositories. The overall goal is to make the VIZZANALYZER framework more suitable for software evolution research. This section describes the goals pursued by this thesis and the criteria used for validating them.

- One goal is the automated source code retrieval of multiple versions from SCM repositories. Before a version can be analyzed, its source code needs to be extracted from a SCM repository and stored locally. Interaction with various repositories is a time-consuming task within software evolution research and a wide number of SCM systems with varying capabilities currently exist. The CONCURRENT VERSIONS SYSTEM (CVS) [7] and SUBVERSION (SVN) [8] are two popular examples. Moreover, only changes within a certain time-span might be interesting and

therefore it is not always necessary to analyze the complete history of a software system. The goal of automated source code retrieval is reached when the implemented solution is able to automatically retrieve the source code of multiple versions of a software system from a CVS or SVN repository without further manual assistance. The user can specify a time-span and a frequency to determine the versions included in the retrieval process. The source code is temporarily stored in a directory on a local hard disk until it has been analyzed. It must be possible to extend the software to support other SCM systems.

- Another goal is the automated analysis of the retrieved source code resulting in graphs. Considering the time-costly retrieval process, it should be possible to invoke a user-defined set of analysis tools on the retrieved versions. Currently, the VIZZANALYZER offers only one low level analysis tool, the RECODER [9], which is able to analyze JAVA source code and construct graphs out of the results. The criterion of this goal is met when the implemented solution is able to automatically invoke the RECODER for each version of the retrieved source code and perform an analysis on it resulting in graphs compatible with VIZZANALYZER. It must be possible to extend the software to support other analysis tools and other programming languages.

- The extraction of evolutional information from SCM repositories is yet another goal. Apart from the mere source code, SCM systems usually record additional metadata associated with the archived data, such as author names or log messages, which could be used for software evolution research. In the following, this metadata will be referred to as evolutional information. This information should be available for further analyses of the generated graphs. To satisfy the criteria of this goal, the implemented solution must be able to automatically extract evolutional information from SCM repositories and attach it to the corresponding graphs that were constructed during the analyses. The information must be attached as properties to the graph and to those nodes of the graph that represent files, classes, or interfaces of the analyzed software system. The structure of the graph itself should not be changed. Moreover, the user must be able to specify which of the properties should be attached.

- In addition, it is a goal to store the resulting graphs into a Database Management System (DBMS). VIZZANALYZER-graphs are memory-intensive data structures. A graph of average size typically consumes several megabytes of system memory. As the implemented solution might generate a high number of graphs, the possibility of serializing runtime instances of graphs and saving them in a storage system is essential to maintain the scalability of the system; i.e., the system should not require significantly more CPU or memory resources for the automated processing of a series of versions than it would require to process a single version. Moreover, saving the results of the retrieval and analysis process is required if these graphs should be utilized by later analyses for software evolution research. The VizzAnalyzer is capable of saving graphs as files and storing them on the local file system but it lacks the ability to store graphs into a DBMS. However, the storage of graphs into a DBMS has several advantages; e.g., it is possible to attach additional information to each graph, making it much easier to identify and manage them. Therefore, the criteria of this goal are met when the implemented solution allows for both automatic and manual storage of the generated graphs into a DBMS and the user can manage and restore the graphs stored inside the DBMS. Furthermore, several types of DBMS should be supported.

- A further goal is the implementation of a graphical user interface (GUI). The VIZZANALYZER comes with a GUI, realized with the JAVA SWING toolkit. It offers an interface to invoke plug-ins as well as a simple generic GUI for high-level analysis plug-ins. However, retrieval plug-ins that require user interaction have to provide their own GUI. To achieve this goal, it is necessary to design and implement a GUI, allowing the user to interact with the implemented solution. More specifically, it must be possible to enter the information required to configure and run the retrieval and analysis process. Furthermore, additional GUIs have to be created for loading and managing graphs stored inside a DBMS as well as saving graphs to a DBMS. During time-intensive operations, status and progress information should be displayed and the user should be able to cancel such operations. If errors occur, they should be displayed as well. In order to be compatible with VIZZANALYZER, the GUI should be realized with the JAVA SWING toolkit.
- Besides these, a general goal of the thesis is a straightforward software design, allowing the implemented components to be extendable and reusable. Of special concern in this respect is the support of additional SCM systems and analysis tools in the future, as these components provide the core capability of the implemented solution. Apart from that, the GUI's architecture has to be designed in a way that allows for easy reuse, while maintaining a certain degree of flexibility for the development of future plug-ins.
- Performance is a secondary goal, as it highly depends on external factors which the implemented solution has no influence on, such as the connection speed to the SCM repository, the efficiency of the graph data structure or the performance of the analysis tool and the DBMS. Nevertheless, frequently executed operations, such as the annotation of graphs with evolutional information or the serialization of graphs, should complete in a feasible time, meaning seconds or minutes rather than hours or days.

The outcome of this thesis is a plug-in for the VIZZANALYZER, allowing it to automatically retrieve and analyze source code from SCM repositories. The results are graphs, annotated with evolutional information, which can be stored in a database. The plug-in fulfills the functional and non-functional requirements defined in Chapter 3. It provides the infrastructure to use the VIZZANALYZER for software evolution research. Further effort is necessary in order to obtain viable results in this field, which is, however, not part of the thesis.

1.4 Context of the Thesis

The VIZZANALYZER has been developed by the School of Mathematics and Systems Engineering (MSI) [10] at Växjö University. It is an ongoing project and the development of the VIZZANALYZER was and is influenced by several student projects and master thesis topics. As such, it was affected to change during the whole developmental period of this thesis. Consequently, the plug-in implemented as part of the thesis had to be adapted when new versions of the VIZZANALYZER introduced changes to its API. If not mentioned otherwise, the text refers to VIZZANALYZER version 1.0.11a, released on November 24, 2006.

1.5 Outline

This section shortly describes the contents of all subsequent chapters of the thesis.

Chapter 2 covers currently existing software tools related to the topic of the thesis. The tools utilized by the implemented solution are introduced.

Chapter 3 describes the features and use cases as well as the functional and non-functional requirements, all following from the goals and criteria stated in Section 1.3.

Chapter 4 introduces architecture and design of the implemented components, which realize the requirements stated in Chapter 3. It explains the task of each implemented component, how they were integrated and how they interact.

Chapter 5 delivers important insights about the implementation of the solution. It mentions noteworthy problems that arose during the development process and how they have been solved.

Chapter 6 concludes the thesis and gives an outlook at future work. It also contains the author’s personal view about the topic of the thesis and its outcome.

2 State of the Art

This chapter discusses existing software tools that are related to the topic of the thesis. Furthermore, the tools on which the implemented solution is based on are introduced.

2.1 Related Work

In this section, available tools related to the topic of the thesis are shortly introduced first and then evaluated to which extent they fulfill the goals of the thesis.

2.1.1 Available Tools

Extracting and analyzing data from SCM repositories is well covered and many tools are available for free. Bevan and Whitehead created IVA [11], a proof-of-concept implementation of their framework for software instability analysis [12]. It extracts data from SVN repositories and builds dependence graphs from JAVA source code. The graphs are analyzed to identify unstable regions and the results are stored in a database. Further user interaction is required to classify and visualize the instabilities.

BLOOF [13] was created by Draheim and Pekacki. It is an infrastructure tool for accessing and analyzing project data from CVS repositories [14]. The data is transformed into a data model and stored in a database. BLOOF realizes interoperation with other tools by separating the data access and analysis layer from the application layer. It provides a JAVA API which other tools can use to perform data access, analysis and visualization.

Alonso, Devanbu and Gertz have presented a framework for the analysis and exploration of software repositories by exploiting database functionality [9]. The goal of this framework is to integrate and manage different documents produced by a development team. Their prototype implementation, called MINERO, uses email messages as a data source. However, the tool is not available for public download.

EROSE [16] is an Eclipse [17] plug-in by Zimmermann et al., which applies data mining to version histories in order to guide programmers along related changes [10]. The resulting association rules are used to predict likely further changes, to detect item coupling, and to prevent errors due to incomplete changes.

Ying et al. have been investigating a similar approach using different data mining algorithms [11]. They conclude that the results reveal dependencies that may not be apparent from other existing analyses. Their tool is not available for public download.

2.1.2 Evaluation

The tools mentioned above are related to data extraction from SCM systems or other types of repositories and the analysis of these data for software evolution research. However, they all exhibit certain deficiencies in reference to the goals and criteria stated in Section 1.3.

Table 2.1 summarizes the features of the introduced tools with respect to the goals of the thesis. It shows that the tools are limited in their usefulness as they only support certain types of repositories or programming languages. The extendibility of their architectures has not been inspected in detail. Only BLOOF mentioned on its project homepage that it was designed to be open for different SCM systems. It can be assumed that IVA is also extendible as it utilizes KENYON which is described in Section 2.3. IVA is the only tool that stores its results into a graph data structure. All tools utilize a DBMS but none of them offers the possibility to directly manage the data stored into the database. The implemented GUIs only allow for basic user interaction. IVA and BLOOF only offer

a GUI for visualizing their results. The retrieval and analysis process has to be invoked on the commando line or from another program. EROSE integrates itself into the GUI of Eclipse. MINERO does not provide a GUI by itself but offers a front-end web application instead for displaying its results in a web browser.

Tool	Supported Repositories	Supported Languages	Graph Data Structure	Database Support		GUI
				Automated Storage	Data Management	
IVA	SVN [(i)]	JAVA	☑	☑	☐	☑[ii]
BLOOF	CVS [i]	flexible	☐	☑	☐	☑[ii]
MINERO	Email Archives	n/a	☐	☑	☐	☐
EROSE	CVS	JAVA	☐	☑	☐	☑
[i]) extendible design, [ii]) only for visualization of results						

Table 2.1: Features of related software evolution research tools

Comparing the introduced tools directly with each other is difficult, as their purposes are rather diverse. IVA's goal is to identify unstable regions in software systems. BLOOF has been designed to analyze and visualize the evolution of software projects. It offers a development kit for creating new analyses on its data model. MINERO can be used to explore and discover information from mailing lists. The purpose of EROSE is to support programmers with recommendations during the development process.

It can be concluded that none of the tools is suitable to satisfy the goals of the thesis. The only tool that can be extended with new analysis is BLOOF. However, it only supports CVS repositories and shows deficiencies with its GUI. Furthermore, the analyses are based on the log history of the repository. It does neither retrieve nor analyze the source code of the inspected system. The other tools address very specific problems while the solution of the thesis should be implemented as a general-purpose tool, providing the infrastructure for covering a wide range of problems within the field of software evolution research.

Hence, a software analysis framework offering basic analysis tools and allowing for the extension with source code retrieval and further analysis tools would fit perfectly to meet the goals and criteria of the thesis. The VIZZANALYZER is such a framework. It is described in the next section.

2.2 VizzAnalyzer

The VIZZANALYZER is a tool for software comprehension. Tools within this field can generally be defined as an abstract software model, views on this model, analyses creating the model, and a mapping between model and view [6]. While the abstract model captures all information required for a certain software comprehension task, the view on this model is creating the image of that information for the humans involved. In order to create the abstract model, a mapping between software and model needs to be defined, known as information extraction, analysis, and focusing. The mapping between model and view is called software visualization. It is possible to separate the domains of software analysis and software visualization by separating model and view. This, in turn, allows the reuse of analysis tools with different visualization back-ends and visualization tools can be reused with different analysis front-ends.

The domain of software analysis can be further distinguished between a base model containing information directly extracted from the software and a final abstract model created by further analyses and focusing of the base model. This distinction increases the reusability even more, as different information extraction front-ends specific to certain source document types can be recombined with different analysis and focusing engines and vice versa.

Another distinction can be made within the domain of software visualization as the same information can be illustrated using different visualization tools. In this case, the view is mapped to a configurable scene before the final image is created, representing a concrete view in terms of the data structure of a particular visualization tool. Distinguishing different scenes for a view increases flexibility and reusability as the same model and view as well as their mappings can be reused by different visualization tools.

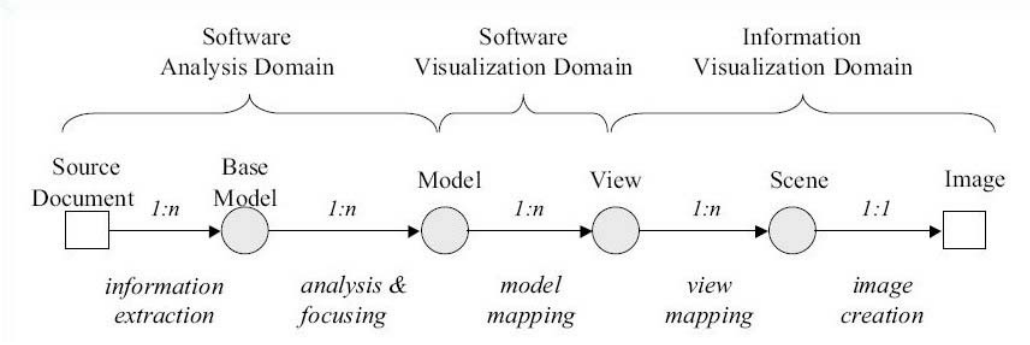

Figure 2.1: Data structures and mappings in software comprehension tools [6]

Figure 2.1 illustrates the different domains within the field of software comprehension as well as the data structures and mappings involved.

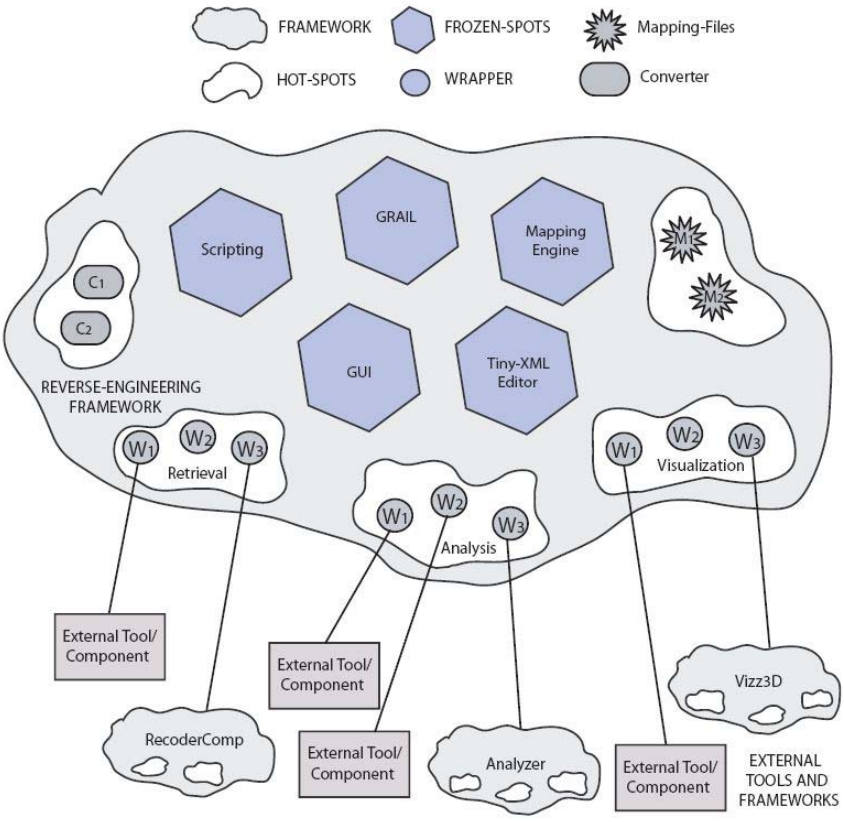

Figure 2.2: Architecture of the VizzAnalyzer framework [12]

The VIZZANALYZER is a framework implementing an architecture which separates data structures as suggested above. The full architecture is actually implemented by two frameworks, VIZZANALYZER and VIZZ3D. However, VIZZ3D is integrated into VIZZANALYZER as a plug-in. It allows configuring the mappings between model, view and scene online instead of programming them. Thereby, it enables the interactive development of software comprehension tools tailored to a particular comprehension context. Figure 2.2 illustrates the architecture of the VIZZANALYZER framework.

2.2.1 Features of VizzAnalyzer

The current version (Release 1.0.11a, 24 Nov. 2006) of VIZZANALYZER includes one information extraction component, one analysis component, and two visualization components. These components are by themselves non-trivial and most of them are available as stand-alone tools. The tools interact within the VIZZANALYZER framework via a general graph data structure which is explained in more detail in the next section.

The component for information extraction is the RECODERCOMP package, based on the RECODER meta-programming library [9] which provides a compiler front-end for JAVA programs and an API for accessing abstract syntax trees (AST) and semantic analysis results. RECODERCOMP produces a set of program representations, such as call graphs, class hierarchies, and containment trees which all can be represented as graphs. Hence, RECODERCOMP is a low-level analysis tool producing the base model discussed before.

In contrast to that, the Analyzer is a high-level analysis component. Using the base model graphs as input, it performs additional calculations in order to recover an architectural level representation of the system under consideration. This is done by computing aggregated information incrementally and filtering out intermediate information irrelevant for the actual comprehension task. Two kinds of high-level analyses are supported: structural analyses and metric analyses. Structural analyses produce new graphs, whereas metric analyses attach metric properties to existing graphs, their nodes, and their edges. For example, the outcome of a high-level analysis could be a class hierarchy graph with nodes that have properties, such as Node Type (`type`), Lines of Code (`LOC`), or Lack of Documentation (`LOD`) attached to them. The values of these properties would look like this: `type="class"`, `LOC="348"` and `LOD="25.0"`, where `type` is a string value indicating whether the node represents a class or an interface, `LOC` is an integer value stating the number of lines of code and `LOD` is a float value representing the percentage of how much documentation is missing.

After additional analyses have been performed on the base model, the final model is mapped to a so called view graph. It is the responsibility of the VIZZANALYZER framework to establish the binding between model and view. This is realized by using XML files which can be configured online within VIZZANALYZER and which contain binding functions specified by the user. The result of this binding is a new graph: the view graph. The structure of this graph is usually copied from the model graph, optionally removing unnecessary elements. The element properties, however, are transformed into new view properties, according to the binding functions specified in the XML file. To extend the example above, the model properties `type`, `LOC` and `LOD` could be transformed into Node Shape (`shape`), Node Height (`height`) and Node Color (`color`), respectively. They would then look like: `shape="box"`, `height="3.48"` and `color="green"`, where `shape` and `color` are string values and `height` is a float value.

Once a view graph has been created, it can be used as input to the visualization components. One of these tools currently available as a plug-in for VIZZANALYZER is YED [21], a graph editor to generate graph drawings and apply automatic layouts to them. It

implements several sophisticated 2D layout algorithms which automatically arrange the elements of a graph. It also supports the user in the process of creating layouts manually.

The other graph drawing tool is VIZZ3D [22] which was originally developed as a component for the VIZZANALYZER but is now also available as a stand-alone tool. Similar to the VIZZANALYZER, VIZZ3D is a reusable framework as well. This gives the user the advantage of reusing all binding functions, metaphors, and layout algorithms. As it is also possible to online-configure visualizations, the appropriate views can be created on demand, making the software analysis process more interactive and iterative. The VIZZ3D framework offers three variation points: binding functions, layouts, and metaphors. Mapping view graphs and their properties to scene graphs is the task of binding functions, which work in the same way as the model bindings of VIZZANALYZER, using online configurable XML files. Layout algorithms are used to arrange the graph elements on the screen by assigning position properties to the nodes of the scene graph. Metaphors could be described as sets of visual objects fitting together in a certain scene or even entities that characterize the environment of the visualization, such as a background image, light sources, or fog. In the example above, a city metaphor could be applied to the class hierarchy graph, displaying nodes as houses and edges as streets. Additionally, the metaphor could display a sky as a background image and a ground on which the virtual city is built on. A hierarchical layout algorithm could be applied to calculate the position of the nodes inside the scene. The binding functions would again transform the properties of the view graph into new properties of the scene graph, such as House Type, House Size, and House Texture. Finally, the user would be able to navigate through the city which actually represents the system under consideration. As a matter of course, the user could choose a less sophisticated metaphor which presents the results of the analyses in a more scientific way; e.g., as a diagram.

2.2.2 Common Data Representation

The VIZZANALYZER framework allows end users to combine and integrate various analysis and visualization tools. As these tools are usually developed independently from each other a mechanism to exchange information has to be provided. Therefore, a common data representation that is general enough to support a variety of data is required. It must be possible to represent the required input and output formats of each tool as an instance of this data structure. Furthermore, a type system is necessary to determine whether a given analysis or visualization is applicable or not [13].

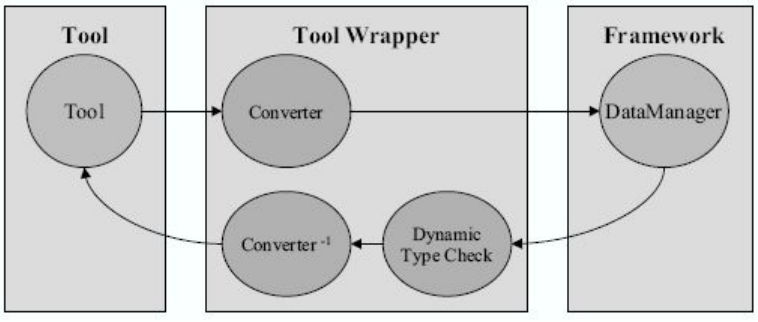

Figure 2.3: Connection between external tools and the framework [13]

To solve these problems the VIZZANALYZER uses annotated graphs where each graph entity (nodes, edges, and the graph itself) has a data object and a set of predicates attached to it. Annotated graphs are general enough to represent almost any kind of data. Another advantage of this data structure is the fact that many frequently used program representations, such as UML diagrams, class hierarchies, or call graphs, can be consi-

dered as graphs. The graph entity predicates constitute a simple, dynamic type system, used by each individual tool to express pre- and post-conditions. These predicates can be checked to find out whether a tool is applicable to a graph. This is the case if the graph's predicates satisfy the preconditions of the tool. The VIZZANALYZER is dynamic in the sense that it does not have a fixed set of predefined predicates.

In order to convert external graph structures created by retrieval or analyses tools to the graph structure of VIZZANALYZER, a wrapper has to be implemented for each tool. Similarly, each visualization tool needs a wrapper to convert VIZZANALYZER's graph structure to its own representation. Another responsibility of wrappers is to perform type checking, using the graph entity predicates mentioned above to detect which graphs a certain tool can use as input. An overview of this concept is given by Figure 2.3. The wrapper mechanism is part of VIZZANALYZER's plug-in architecture presented in the next section.

2.2.3 Plug-In Architecture

The VIZZANALYZER contains three variation points that make it possible to plug in external tools into the framework, as shown in Figure 2.4. These are information extraction, analysis, and visualization.

Technically, the variation points are organized as directories containing wrapper classes which extend predefined interfaces. These classes are dynamically read in when VIZZANALYZER starts to make the corresponding plug-ins available at runtime.

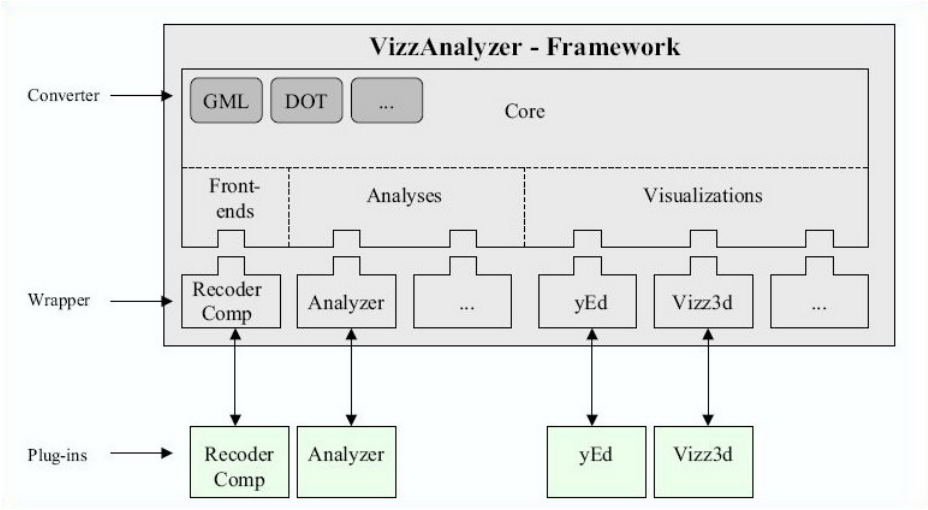

Figure 2.4: Plug-in architecture of the VizzAnalyzer framework [6]

The wrapper classes delegate method calls from VIZZANALYZER to the plug-ins and vice versa. They also check if the input-data fulfills the preconditions required to run a certain plug-in. Moreover, it is their responsibility to perform data adaptation between the graph data structure of VIZZANALYZER and the data format of the plug-in. For this purpose, the VIZZANALYZER contains a collection of conversion adapters for standardized formats, such as GML [24] or DOT [25], which are reusable by any wrapper. If necessary, more converters can be added to this collection.

2.3 Kenyon

KENYON [26] is an open-source data retrieval, preprocessing and storage backend designed to facilitate software evolution research. It aims to assist researchers of this field

by reducing the manual labor costs for these time-expensive operations as much as possible.

Currently, the most labor-intensive tasks in software evolution research are those of interacting with various data archives (e.g., SCM systems), identifying and extracting semantically consistent configurations, and analysis-tool invocation [15]. Many research tools [7; 8; 9; 10; 11] sample the archived project artifacts, such as source code configurations, at specific points in time over the entire project history or over certain parts of it. During this process, different types of facts are extracted from each configuration or from the data associated with the difference between pairs of configurations. These facts serve as input data on which the evolution analysis operates. Commonly, they are stored only once and reused during multiple analysis passes.

When a new software evolution research system is created, the method of retrieving input data, extracting relevant facts and storing results is often implemented from scratch and researchers are forced to make certain tradeoffs due to the time cost of the implementation. For example, the type of data that can be analyzed is limited or the system supports only a single SCM system.

It follows that any assistance tool for software evolution research should be capable of automating the fact extraction process to minimize the manual labor a researcher would have to invest otherwise. Another concern is that new research systems should not need to reinvent solutions to the common data retrieval, preprocessing and storage problems. Furthermore, the assistance tool should not limit the types of evolution research analyses which are applicable to the preprocessing results.

KENYON was designed to fulfill these requirements. Since it is an open-source JAVA tool, a linkage between KENYON and the VIZZANALYZER lies at hand. Integrating KENYON into the VIZZANALYZER as a plug-in tool should provide the necessary functionality to solve the problem of the thesis. The next two sections describe KENYON's features and architecture in more detail.

2.3.1 Features of Kenyon

KENYON provides automated source code retrieval from SCM repositories onto the local file system, invokes analysis-specific Fact Extractors on each retrieved version, and optionally saves the extracted facts into a relational database using an object-relational mapping (ORM) system. The data structures of the ORM system provide a common basis for interpreting and reusing these results as they can be accessed by any evolution analysis tool. However, KENYON does not mandate analysis tools to report results in a specific format.

KENYON was designed to require as little user-interaction during runtime as possible. Hence, the user has to supply a configuration file that specifies the source SCM repository, version extraction guides, and the external analysis tools to be invoked on each retrieved version. Another configuration file is necessary to specify the database into which the preprocessed data should be stored as well as other ORM-specific properties.

SCM repositories are "sampled" at a specified time interval, such as once per second or twice per day, between a start date and an end date, which may be set to "last" for incremental processing. The current version of KENYON (Release 1.3.1, April 18 2005) supports the CVS [7], SUBVERSION [8], and CLEARCASE [28] SCM systems. It is also possible to process pre-downloaded versions, in case when access to a SCM repository is not available.

2.3.2 Architecture

The high-level data flow architecture of KENYON is shown in Figure 2.5. The numbers on the solid arrows indicate the processing order. The class `DataManager` is the entry point of the execution. It reads the configuration files and invokes the source code retrieval, fact extraction, and object storage methods. The class `SCMInterface` isolates KENYON from the implementations of each concrete SCM subclass. The abstract classes `FactExtractor` and `MetricLoader` serve as variation points for external tool invocation extensions.

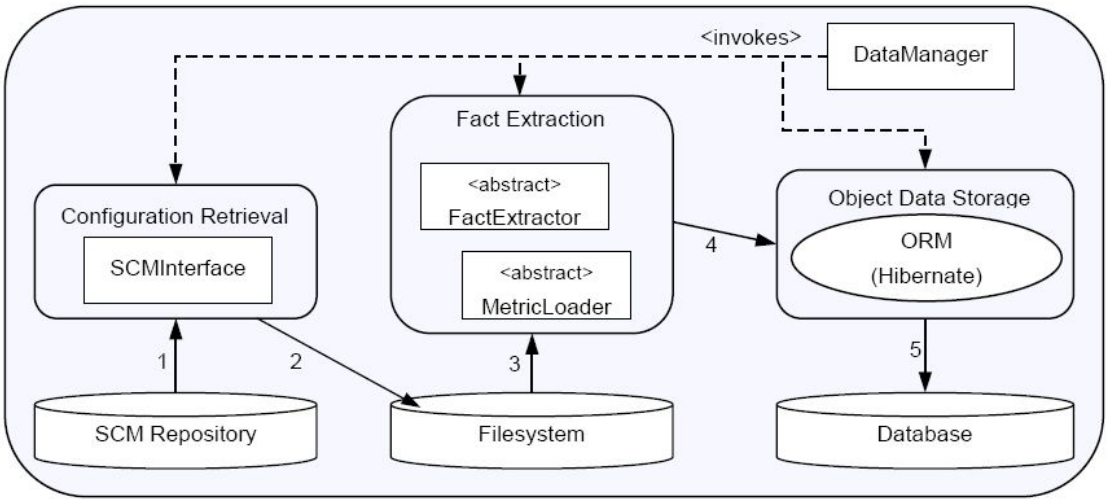

Figure 2.5: High-level data flow architecture of Kenyon [15]

KENYON retrieves each version to be processed and places it in the local file system. The `DataManager` class then invokes the series of concrete `FactExtractor` and `MetricLoader` subclasses specified by the user in the configuration file. These subclasses are the means by which external analysis tools interface with KENYON. Hence, they have to be implemented by the user.

KENYON supports both low-level and high-level analysis tools (cf. Section 2.2). The former tools are the Fact Extractors, mentioned before, while the latter are termed Metric Loaders. The Fact Extractors create a number of base models serving as input for the Metric Loaders which perform further analyses resulting in the final models.

The results from each processed version can optionally be stored into a database. KENYON utilizes HIBERNATE, an object-relational mapping system, to help automate the storage and retrieval of JAVA objects to and from the database [16].

3 Requirements

This section describes the users, the features, and the use cases of the software as well as the resulting functional and non-functional requirements. It also explains which constraints apply and which assumptions were made for the implementation. The software components developed as part of the thesis have to meet these requirements in order to accomplish the goals and criteria of Section 1.3.

3.1 Users

There is only one user group expected for the implemented components: users who work with the VIZZANALYZER. They want to utilize these components in order to automatically analyze one or more versions of a software system which source code is retained inside a SCM repository. The user expects a number of graphs as results, annotated with evolutional information, which can optionally be written to a DBMS. All users have the necessary skill to set-up and use the DBMS and the SCM systems they want to work with.

3.2 Features

This section describes the features of the implemented solution. The features realize the goals and criteria of the thesis defined in Section 1.3.

Feature F01 Retrieve source code from SCM repositories

Description: It is possible to automatically retrieve source code of multiple versions from CVS and SVN repositories. The source code is temporarily stored in a directory on the local file system until it has been analyzed. The user is able to specify a configuration file containing all parameters necessary to access the SCM repository, including a time-span and a frequency which determine the versions to be retrieved.

Feature F02 Analyze source code and generate graphs

Description: It is possible to automatically analyze the retrieved source code, resulting in graphs. After a version has been retrieved and stored, the RECODER tool will be invoked to analyze it. This step results in a number of graphs created by the RECODER tool, which are temporarily stored inside VIZZANALYZER's internal graph storage data structure. The user can specify a configuration file with the extension `".vap"`, containing all parameters necessary to run the RECODER tool. It is also possible to choose which of the basic graph types should be generated. Optionally, other graph types can be specified in a configuration file with the extension `".xml"`.

Feature F03 Extract evolutional information from SCM repositories

Description: It is possible to extract evolutional information from CVS and SVN repositories. The information will be attached as properties to the corresponding graphs created during the analyses and to those nodes of the graph that represent files, classes, or interfaces. The properties attached to the graphs are: Branch Tag and Commit Date. The properties

attached to the nodes are: Number of Check-ins, Last Check-in Date, Revision Number, Last Log Message, Author Names, Number of Authors, and Last Author. The user is able to specify which of the properties should be attached, if they should be determined in an absolute or relative manner, and whether the first version should be treated differently.

Feature F04 Store graphs into a DBMS

Description: It is possible to store graphs into a Database Management System. The graphs resulting from the source code retrieval and analysis process are automatically written into the DBMS. The user can also manually save any other graph available in VIZZANALYZER's user interface to the DBMS. Graphs stored inside a database are categorized into projects and the user is able to choose an existing project or specify a new project name. It is also possible to specify a short description for the graphs. These descriptions can be generated automatically for graphs resulting from the source code retrieval and analysis process. The access to the DBMS is specified in a configuration file. All common types of relational DBMS are supported.

Feature F05 Load graphs from a DBMS

Description: It is possible to load graphs that have been written into a DBMS to the internal graph storage of VIZZANALYZER. The user is able to browse through a list of existing projects and select one or more graphs that should be loaded. The list displays the description, the date of creation and last modification, the number of nodes and edges as well as the attached properties of each graph.

Feature F06 Managing graphs stored inside a DBMS

Description: It is possible to manage the graphs stored inside a DBMS. The user is able to browse through a list of existing projects and graphs and select one or more of them. The list displays the description, the date of creation and last modification, the number of nodes and edges as well as the attached properties of each graph. It is possible to perform the following operations: create, rename, and delete projects, edit graph descriptions, copy or move graphs between projects, and delete graphs.

Feature F07 Graphical user interface

Description: Another feature is the graphical user interface. It provides a dialog window to enter the information required to configure and run the retrieval and analysis process. Furthermore, there are different dialog windows for loading and managing graphs stored inside a DBMS as well as saving graphs to a DBMS. During time-intensive operations, status and progress information are displayed in a window and the user is able to cancel such operations. If errors occur, they are displayed in a window as well.

Feature F08 Capability for extensions

Description: The implemented solution can be extended to support additional SCM systems and future analysis tools.

Feature F09 Reusability

Description: It is possible to reuse basic parts of the GUI for the development of future plug-ins.

Feature F10 Performance

Description: Operations that do not rely on external factors can complete in a feasible time, meaning seconds or minutes rather than hours or days.

3.3 Use Cases

This section describes the use cases that provide the basis for the formulation of the functional requirements. The use cases are illustrating the features to make them more vivid and easier transformable into requirements. Moreover, they describe how the user interacts with the system.

Each use case has a particular task. It contains a title, a brief description, a list of actors involved, the preconditions that have to be fulfilled before the use case can start as well as the condition of the system after the use case has ended. It is also listed which features are covered by which use case. The basic path represents the regular flow of events for the use case, but there can also be several alternative and/or exception paths for each particular task. The first numbering value of these paths starts at the step where they leave the basic path. Each path is described by a short title.

Use Case UC1 Retrieve and analyze source code Feature#: F01, F02, F03, F04

Description: This use case describes how the process of source code retrieval and analysis from a SCM repository takes place. First, a dialog window appears where the user specifies the location of the configuration files for RECODER and KENYON as well as a number of user-options. When the user clicks on the button that starts the process, a new window opens showing progress and status information of the retrieval and analysis process. The user is able to cancel the running process. When the process is finished, the window can be closed by clicking the corresponding button.

Actors: The user.

Preconditions: The preconditions to this use case are that VIZZANALYZER has started and is ready for user interaction. The system has access to the repository from which the source code should be retrieved. The necessary command-line clients for KENYON are installed and accessible. If the graphs which are generated should be written to a database, the corresponding DBMS is running and accessible. A JAVA driver for the DMBS is available and the configuration file for the ORM system has been set up.

Post Conditions: The post conditions are that source code has been retrieved and analyzed according to the specified settings and that a number of graphs have been generated and annotated with the specified evolutional information. The graphs have either been passed on to the internal graph storage of VIZZANALYZER and are now visible in its user interface and/or they have been written to the database.

Trigger: The user selects the corresponding menu entry for the Kenyon plug-in from VIZZANALYZER's user interface.

Basic Path: **Source code retrieval and analysis**

1. The system displays a dialog window, containing setup options for the retrieval and analysis process.
2. The user has to specify the setup options of the dialog. These consist of:
   - the location of the configuration files for KENYON and RECODER, and the location of an optional configuration file for generating custom graph types,
   - the default graph types that should be generated (i.e., Default LLA Tree graph, Class Hierarchy graph, and Reference graph),
   - the properties that should be attached to the graphs and the method for their calculation (cf. Feature F03),
   - the storage method (i.e., VIZZANALYZER's internal graph storage and/or database), and
   - the database parameters (i.e., project name and graph description).
3. The user clicks on the button that starts the retrieval process.
4. The system checks if the settings are valid.
5. The system starts the retrieval and analysis process. The dialog window closes and a progress window opens, showing the current status of the process.
6. The system executes the process.
7. The system has finished the process and displays a corresponding status message in the progress window.
8. The user clicks on the button that closes the progress window. The window closes and the use case ends.

Alternative Path A: **User cancels process**

7. The user clicks on the button that cancels the process.
8. The system stops the process and displays a corresponding status message in the progress window.
9. The user clicks on the button that closes the progress window. The window closes and the use case ends.

Exception Path I: **Invalid dialog settings**

5. The system recognizes invalid settings.
6. A message window informs the user that the settings are invalid.
7. Go to Step 2 (Basic Path).

Exception Path II: **Recoder fails to analyze source code**

7. RECODER failed to analyze the current version.
8. A question window opens, asking the user how to proceed.
9. The user chooses one of the following options:
   a) **Ignore:** Go to Step 6 (Basic Path).
   b) **New configuration file:** The user specifies a new configuration file for RECODER. Go to Step 6 (Basic Path).
   c) **Terminate:** Go to Step 8 of Alternative Path A.

Use Case UC2 Load graphs from database Feature#: F05

Description: This use case describes how graphs are loaded from a database into the VIZZANALYZER. The user is able to browse through all graphs that are stored inside the database. Several graphs from different projects can be loaded at the same time.

Actors: The user.

Preconditions: The preconditions to this use case are that VIZZANALYZER has started and is ready for user interaction. The DBMS from which the graphs should be loaded is running and accessible. A JAVA driver for the DMBS is available. The configuration file for the ORM system has been set up.

Post Conditions: The post condition is that the graphs selected by the user have been passed on to the internal graph storage of VIZZANALYZER and are now visible in its user interface.

Trigger: The user selects the corresponding menu entry for this action from VIZZANALYZER's user interface.

Basic Path: **Load graphs from database**

1. The system displays a window that allows the user to browse through all projects and graphs stored inside the database.
2. The user selects one or more projects. The graphs contained inside the selected projects are listed.
3. The user selects one or more graphs. The attached properties of the last selected graph are displayed.
4. The user clicks on the button that loads the selected graphs.
5. The system starts to load the graphs. The window switches to a progress window, showing the current status of the process.
6. The system loads the graphs from the database and writes them into the internal graph storage of VIZZANALYZER.
7. The system has finished the process. The progress window closes and the use case ends.

Alternative Path A: **User cancels process**

7. The user clicks on the button that cancels the process.
8. The system stops the process and displays a corresponding status

message in the progress window.
9. The window closes and the use case ends.

Use Case UC3 Manage graphs inside a database Feature#: F06

Description: This use case describes how the graphs stored inside a database are managed. The user is able to rename, delete, or create new projects. It is also possible to edit the description of graphs, to delete graphs, and to copy or move graphs from one project to another.

Actors: The user.

Preconditions: The preconditions to this use case are that VIZZANALYZER has started and is ready for user interaction. The DBMS from which the graphs should be loaded is running and accessible. A JAVA driver for the DMBS is available. The configuration file for the ORM system has been set up.

Post Conditions: The post condition is that the projects or graphs have been modified according to the user's actions. The graphs stored inside the internal graph storage of VIZZANALYZER are not affected.

Trigger: The user selects the corresponding menu entry for this action from VIZZANALYZER's user interface.

Basic Path: **Managing graphs**

1. The system displays a window that allows the user to browse through all projects and graphs stored inside the database.
2. The user takes one of the following actions:
   a) **New project:** Go to Alternate Path A.
   b) **Rename project:** Go to Alternate Path B.
   c) **Delete project:** Go to Alternate Path C.
   d) **Edit graph description:** Go to Alternate Path D.
   e) **Copy graphs:** Go to Alternate Path E.
   f) **Move graphs:** Go to Alternate Path F.
   g) **Delete graphs:** Go to Alternate Path G.
   h) **Close window:** Go to Step 3 (Basic Path).
3. The user clicks on the button that closes the window. The window closes and the use case ends.

Alternative Path A: **Creating a new project**

3. The user clicks on the button that creates a new project.
4. A dialog window opens, asking for the name of the new project.
5. The user enters the name.
6. The system creates the new project in the database and the window closes.
7. Go to Step 2 (Basic Path).

Alternative Path B:

Renaming a project

3. The user selects a project name from the list of projects.
4. The user clicks on the button that renames a project.
5. A dialog window opens, asking for the new name of the project.
6. The user enters the name.
7. The system changes the project name in the database and the window closes.
8. Go to Step 2 (Basic Path).

Alternative Path C:

Deleting projects

3. The user selects one or more project names from the list of projects.
4. The user clicks on the button that deletes projects.
5. A dialog window opens, asking the user to confirm.
6. The user confirms.
7. A progress window opens, showing the current status of the process.
8. The system deletes the selected projects and their graphs from the database.
9. The progress window closes.

Go to Step 2 (Basic Path).

Alternative Path D:

Editing a graph description

3. The user selects a graph from the list of graphs.
4. The user clicks on the button that edits the description of a graph.
5. A dialog window opens, asking for the new description.
6. The user enters the description.
7. The system changes the description in the database and the window closes.
8. Go to Step 2 (Basic Path).

Alternative Path E:

Copying graphs

3. The user selects one or more graphs from the list of graphs.
4. The user clicks on the button that copies graphs.
5. A dialog window opens, asking the user for the project name to which the graphs should be copied.
6. The user selects a project name.
7. A progress window opens, showing the current status of the process.
8. The system copies the graphs to the selected project.
9. The progress window closes.

Go to Step 2 (Basic Path).

Alternative Path F:

Moving graphs

3. The user selects one or more graphs from the list of graphs.
4. The user clicks on the button that moves graphs.
5. A dialog window opens, asking the user for the project name to which the graphs should be moved.
6. The user selects a project name.
7. A progress window opens, showing the current status of the process.
8. The system moves the graphs to the selected project.
9. The progress window closes.

Go to Step 2 (Basic Path).

Alternative Path G:

Deleting graphs

3. The user selects one or more graphs from the list of graphs.
4. The user clicks on the button that deletes graphs.
5. A dialog window opens, asking the user to confirm.
6. The user confirms.
7. A progress window opens, showing the current status of the process.
8. The system deletes the selected graphs.
9. The progress window closes.

Go to Step 2 (Basic Path).

Alternative Path H:

User cancels process (applies to alternative paths C, E, F and G)

9. The user clicks on the button that cancels the process.
10. The system stops the process and displays a corresponding status message in the progress window.
11. The window closes.
12. Go to Step 2 (Basic Path).

Use Case UC4 Store graphs into database Feature#: F04

Description: This use case describes how graphs that are available in VIZZANALYZER are stored into a database. It is possible to write all graphs in one single step or to select one or more graphs that should be written from the internal graph storage of VIZZANALYZER to the database.

Actors: The user.

Preconditions: The preconditions to this use case are that VIZZANALYZER has started and is ready for user interaction. The DBMS from which the graphs should be loaded is running and accessible. A JAVA driver for the

DMBS is available. The configuration file for the ORM system has been set up.

Post Conditions: The post condition is that the graphs which the user wants to store have been written to the database.

Trigger: The user selects the corresponding menu entry for this action from VIZZANALYZER's user interface.

Basic Path: **Store graphs into database**

1. The system displays a dialog window, asking the user for a project name and an optional graph description.
2. The user selects a project name and enters a graph description.
3. The user clicks on the button that saves the graphs.
4. The system starts to save the graphs. The window switches to a progress window, showing the current status of the process.
5. The system saves the graphs to the database.
6. The system has finished the process. The progress window closes and the use case ends.

Alternative Path A: **Overwrite existing graph**

6. The system detects that a graph already exists in the database.
7. A question window opens, asking the user how to proceed.
8. The user chooses one of the following options:
   a) **Overwrite (new description):** The user enters a new description and the system overwrites the graph. Go to Step 5 (Basic Path).
   b) **Overwrite (keep description):** The system overwrites the graph. Go to Step 5 (Basic Path).
   c) **Save as new graph:** The user enters a new description and the system saves the graph. Go to Step 5 (Basic Path).
   d) **Do not save graph:** The graph is not saved. Go to Step 5 (Basic Path).

Alternative Path B: **User cancels process**

6. The user clicks on the button that cancels the process.
7. The system stops the process and displays a corresponding status message in the progress window.
8. The window closes and the use case ends.

3.4 Functional Requirements

This section describes the functional requirements for the implementation of the software components. The requirements specify what these components shall be able to do and are a result from the features and use cases described above. A unique number is given to each requirement for reference purpose as well as a descriptive short name. Moreover, each requirement includes a reference to the corresponding use case, a description, a rationale, and a fit criterion.

Requirement R01 Specify configuration files **Use Case#: UC1**

Description: The user shall be able to specify the location of the configuration files for KENYON and RECODER.

Rationale: Both KENYON and RECODER read project-related parameters from a file when they start. It is reasonable that the user can specify the location to these configuration files rather than to enter the necessary parameters into a dialog window each time the application runs. Thus, it is possible to reuse previously created configuration files for specific projects.

Fit Criterion: The user is able to enter the file paths manually or to select the configuration files via a file-chooser dialog.

Requirement R02 Verify file locations **Use Case#: UC1**

Description: The system shall be able to verify the specified file locations.

Rationale: If the specified file locations are incorrect, the process will produce errors at a later point.

Fit Criterion: The system checks if the specified files exist and are readable before the process starts. It displays an error message if this is not the case.

Requirement R03 Set Recoder options **Use Case#: UC1**

Description: The user shall be able to set options that affect the Recoder-run.

Rationale: It is possible that the RECODER fails to analyze the source code retrieved from the SCM repository. Several actions can be taken in such a case to continue the overall retrieval and analysis process, which the user should be able to specify. It is also possible that RECODER requires a different configuration file for each version of the retrieved source code in order to analyze it.

Fit Criterion: The user is able to set the following options:

- A default action when the Recoder-run fails: The user can select one of the following four actions: Ask user for a new configuration file (1), Ignore this build (2), Terminate Kenyon-run (3), or Ask user what to do (4).
- Always ask the user for new a configuration file before RECODER runs: The user can check or uncheck this.

These options are used when RECODER runs to analyze source code. If the user is asked for a new configuration file, the system displays a file-chooser dialog. If the user is asked what to do, a dialog window with the following three options is displayed: Specify a new configuration file (1), Ignore this build (2), and Terminate Kenyon-run (3).

Requirement R04 Specify output graph types **Use Case#: UC1**

Description: The user shall be able to specify which types of graphs are created.

Rationale: Generating graphs can take up a huge amount of time and memory, depending on the analyzed software. By default, RECODER creates three graph types at each run. However, the configuration file of RECODER does not allow for changing this setting. Neither is it possible to specify other graph types than the three default types.

Fit Criterion: The user is able to specify which of the three default graph types are created. These are: Default LLA Tree graph (1), Class Hierarchy graph (2), and Reference graph (3). Optionally, the location of a graph configuration file which provides the necessary information for creating other graph types can be specified.

Requirement R05 Specify graph and node properties **Use Case#: UC1**

Description: The user shall be able to specify which graph and node properties should be attached to the generated graphs.

Rationale: The user might only be interested in certain properties. Hence, it should be possible to choose which properties should be attached.

Fit Criterion: The user is able to select which properties should be attached to generated graphs and their nodes. These properties are described in Feature F03.

Requirement R06 Specify relevant graph and node types **Use Case#: UC1**

Description: The user shall be able to specify which graph and node types are relevant for the annotation process.

Rationale: Attaching properties can take up a considerable amount of time, as the system has to match file names of the SCM repository with file, class, or interface names of nodes in order to find the node to which the information should be attached to. However, graphs like the Default LLA Tree graph might contain many irrelevant nodes to which no properties can be attached. Therefore, the annotation process will complete faster if only relevant graph and node types are considered.

Fit Criterion: The user is able to specify a comma-separated list of graph and node types that are considered during the annotation process.

Requirement R07 Set property value options **Use Case#: UC1**

Description: The user shall be able to set property value options.

Rationale: The values of some node properties can be determined in an absolute or relative manner. This means that they are determined either by considering all previous versions (absolute) or by considering only the version that was analyzed right before the current one (relative). It should be possible to treat the first analyzed version differently, since it is not necessarily the first version in the SCM repository. The three

node properties that are affected are: Number of Check-ins (1), Author Names (2), and Number of Authors (3).

Fit Criterion: The user is able to specify whether the property values should be determined in an absolute or relative manner. When the method is set to relative, the user can select one of the following three for property values of the first version: Absolute (1), Relative (2), or Do not create properties (3).

Requirement R08 Set graph storage options Use Case#: UC1

Description: The user shall be able to specify whether the generated graphs should be written to the database and/or to the internal graph storage of VIZZANALYZER.

Rationale: If the user expects only a small number of graphs to be generated it should be possible to keep them in the internal graph storage of VIZZANALYZER, whereas a larger number of graphs should not be kept in memory but written to a database instead.

Fit Criterion: When the database is available, the user is able to select one of the following three options: Do not write to database (1), Write to both memory and database (2), or Write to database only (3).

Requirement R09 Specify database storage parameters Use Case#: UC1, UC4

Description: The user shall be able to specify a project name and a description when graphs should be written to a database.

Rationale: Graphs stored in a database are categorized into projects and may have a description to make it easier to identify them. Therefore, it must be possible to specify the project name and the description.

Fit Criterion: The user is able to select an existing project name or to enter a new name. It is also possible to enter a graph description.

Requirement R10 Auto-generate graph descriptions Use Case#: UC1

Description: The system shall be able to generate graph descriptions automatically when graphs are written to database during the retrieval and analysis process.

Rationale: The graph description is an important attribute, as it helps the user to identify the graphs stored in a database. It should contain enough information to make it unique to other graphs belonging to the same project. Therefore, it should be possible to generate graph descriptions automatically using information stored in the SCM repository as well as project information.

Fit Criterion: The user is able to choose whether graph descriptions should be generated automatically or not. Auto-generated descriptions have the following format:

```
[last check-in date] :: [project name] :: [graph label]
```

Requirement R11 Source code retrieval **Use Case#: UC1**

Description: The system shall be able to retrieve multiple versions of source code within a specified time-span and in a specified frequency from a CVS or SVN repository, and to store this source code in a temporary directory on the local file system.

Rationale: Retrieving source code from SCM repositories is necessary to provide software evolution research capabilities for VIZZANALYZER.

Fit Criterion: The system is able to invoke KENYON with all necessary parameters, which offers the capability to retrieve source code from CVS and SVN repositories.

Requirement R12 Source code analysis and graph generation **Use Case#: UC1**

Description: The system shall be able to analyze the source code retrieved from SCM repositories and to create graphs that can be read by VIZZANALYZER.

Rationale: The source code has to be transformed into the common data representation of VIZZANALYZER (cf. Section 2.2.2) in order to be reusable by other analysis or visualization tools of the VIZZANALYZER framework.

Fit Criterion: The system is able to invoke RECODER, offering the capability to analyze source code and to create graphs, with all necessary parameters.

Requirement R13 Graph annotation with evolutional information **Use Case#: UC1**

Description: The system shall be able to annotate the generated graphs with evolutional information from the SCM repository.

Rationale: Annotating the generated graphs with evolutional information is necessary to provide software evolution research capabilities for VIZZANALYZER.

Fit Criterion: The system is able to access the log history extracted by KENYON from the SCM repository and to attach parts of this information as properties to the generated graphs and their nodes. The properties are described in Feature F03.

Requirement R14 Automated graph storage **Use Case#: UC1**

Description: The system shall be able to store graphs automatically into a database

during the retrieval and analysis process.

Rationale: Depending on the size of the analyzed software system, the corresponding graphs can take up a considerable amount of memory. If many versions are analyzed, the system could run out of physical memory, which has great impact on the system's performance. To avoid this situation, it should be possible to write graphs to a database instead of holding them all in local memory. Furthermore, the retrieval and analysis process is a time-costly task and thus, the results should be available for later analyses.

Fit Criterion: The system is able to store graphs into a database in such a way that it is possible to fully restore them. Graphs stored inside a database are categorized into projects and can contain a description. All common types of relational database management systems are supported if the corresponding driver is available.

Requirement R15 Restore graphs from a DBMS Use Case#: UC2

Description: The system shall be able to restore graphs stored inside a database and to write them into the internal graph storage of VIZZANALYZER.

Rationale: Graphs written to a database must be restored to become available inside VIZZANALYZER for further analyses.

Fit Criterion: The system is able to fully restore graphs stored inside a database and to write them into the internal graph storage of VIZZANALYZER.

Requirement R16 Load graphs from a DBMS Use Case#: UC2

Description: The user shall be able to browse through the list of all graphs stored inside a database and to select the graphs that should be loaded into VIZZANALYZER.

Rationale: The user must be able to load graphs that have been written to a database, in order to use them in VIZZANALYZER.

Fit Criterion: The user is able to load graphs from a database into VIZZANALYZER as described in Use Case UC2.

Requirement R17 Manage graphs and projects inside a DBMS Use Case#: UC3

Description: The user shall be able to manage all projects and graphs stored inside a database.

Rationale: The user must be able to manage projects and graphs stored inside a database.

Fit Criterion: The user is able to manage projects and graphs stored inside a database as described in Use Case UC3.

Requirement R18 Manual graph storage **Use Case#: UC4**

Description: The user shall be able to manually store graphs available in VIZZANALYZER's user interface into a database.

Rationale: Graphs generated during the retrieval and analysis process can be written automatically into a database. However, there should also be the possibility to select graphs manually from VIZZANALYZER's user interface and store them into a database.

Fit Criterion: The user is able to select graphs from the VIZZANALYZER interface, which should be stored into a database. It is also possible to store all available graphs in a single step into a database.

Requirement R19 Detect DBMS availability **Use Case#: UC1 – UC4**

Description: The system shall be able to detect whether the configured database is accessible or not.

Rationale: If the database is not available, the user should be informed before any database operation starts, in order to avoid error messages.

Fit Criterion: The system is able to probe the database for its current status.

Requirement R20 Display progress information **Use Case#: UC1 – UC4**

Description: The system shall be able to display a progress window for operations that could be time intensive, offering the possibility to cancel the operation.

Rationale: The user wants to know about the progress of active operations. It should be possible to cancel such operations if necessary.

Fit Criterion: The system is able to display a progress window containing a button that cancels the process for operations that could be time-intensive. Operations can be stopped when the user clicks on this button.

3.5 Non-Functional Requirements

This section explains which non-functional requirements apply to the implemented solution. Non-functional requirements specify criteria in respect to the operation of a system, rather than specific behaviors or functions. Reliability and efficiency are typical non-functional requirements.

Performance

The performance requirement applies to all operations that do not rely on external factors, such as internet connection speed, database performance, or third-party components. These operations shall complete in a feasible time, meaning seconds or minutes rather than hours or days.

Reusability

This requirement applies in particular to the database functionality and the GUI. It shall be possible to reuse the capability of writing graphs from the internal graph storage of

VIZZANALYZER to a database and vice versa. Furthermore, basic parts of the GUI shall be reusable for the development of future VIZZANALYZER plug-ins.

Extendibility

The implemented solution shall be extendable to support additional SCM systems and future analysis tools.

Operational Requirements

As part of the VIZZANALYZER framework, the implemented solution shall run on any system that is supported by VIZZANALYZER. However, additional requirements that do not apply to VIZZANALYZER, such as internet access, database driver or SCM clients might limit the operation on certain systems.

3.6 Constraints

This section describes constraints applying to the design and implementation of the software components realizing the requirements listed above. Some constraints determine the usage of a certain technology, which is described as well.

VizzAnalyzer Plug-In

It is a constraint from the thesis description that the solution should be implemented as a plug-in for VIZZANALYZER. As VIZZANALYZER is implemented in JAVA using SWING components, this constraint determines the implementation language of the solution.

Recoder

The usage of RECODER is an implicit constraint, as it currently is the only low-level analysis tool (i.e., graph generation) that is available for the VIZZANALYZER. RECODER is able to analyze JAVA source code.

Kenyon

Although not a constraint by means, the usage of KENYON was encouraged by the thesis description. KENYON offers support for several types of SCM repositories. It is implemented in JAVA and available as open source.

Hibernate

The usage of HIBERNATE is another constraint which does not originate from the thesis description. However, HIBERNATE simplifies the usage of relational database management systems for JAVA developers as it provides a powerful and completely object-oriented interface. It supports all common types of relational databases and is available as open source as well.

3.7 Summary

This chapter is essential for the outcome of the thesis. The capabilities of the developed software components are precisely defined, and hence, it is the basis for their architecture and implementation. The goals and criteria defined in the introduction chapter lead to the features stated here. These features are then illustrated by use cases and transformed into the requirements.

Since each use case references to its corresponding features and each functional requirement references to its corresponding use cases, it is easy to see the connection between them. The architecture described in the following has to incorporate all these requirements to meet the criteria of the stated goals.

4 Outline of the Solution

This chapter describes architecture and design of the implemented components realizing the requirements stated in Chapter 3. First, it defines the process constituting the main goal of the thesis. Secondly, it explains which components were implemented, how they were integrated into the VIZZANALYZER and how they interact. It then takes a closer look at each of these components and finally concludes with a summary of the chapter.

The UML class diagrams in this chapter are intended to give a high-level structural overview of the implemented or otherwise involved components. They do not precisely reflect the implemented classes and are sometimes incomplete. The attributes and methods of classes are only listed when they are necessary for understanding architectural details or when they are of importance in other respect. The same applies to input and return parameters of methods. Generally, class names are not fully qualified; i.e., package names are omitted.

4.1 Retrieval and Analysis Process

The aim of the thesis is to extend the VIZZANALYZER with the ability to automatically retrieve and analyze source code and evolutional information from SCM repositories. Implementing this process, called the Retrieval and Analysis Process, as a plug-in for the VIZZANALYZER is therefore the main goal. The process is part of Use Case UC1, described in Section 3.3. This section defines the process by explaining its elements and its flow.

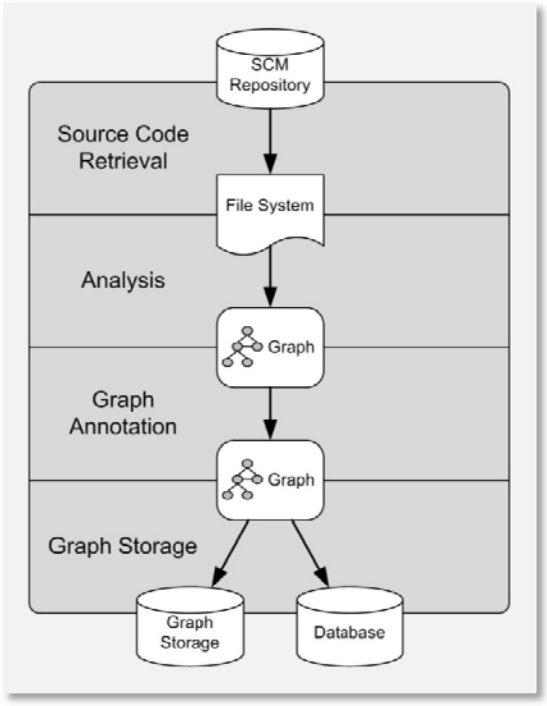

Figure 4.1: The Retrieval and Analysis Process

The Retrieval and Analysis Process can be decomposed into four steps. First, during the Source Code Retrieval process (1) the SCM repository is accessed to retrieve the source code of the current version. The source code is temporarily stored on the local file system. The Analysis process (2) is then responsible for transforming the retrieved source code into graphs which can be stored in the internal graph storage of VIZZANALYZER and accessed by other tools (cf. Section 2.2.2). At the next step, Graph Annotation (3), evolutional information extracted from the SCM repository is attached to the graphs

created during the Analysis process. Most of the information is attached to those nodes of the graphs that represent classes, interfaces or files. The final step is called Graph Storage (4), which stores the graphs into the database and/or the internal graph storage of VIZZANALYZER, according to the user settings. The process is illustrated by Figure 4.1.

The Retrieval and Analysis Process is iterative, which means that it is repeated until the last version has been retrieved and processed. The versions that should be retrieved from the SCM repository are not selected directly by the user. Rather, the user specifies a start and an end time between which the versions of interest have been checked-in into the repository. Additionally, a time interval determines which versions between this time-span should be retrieved.

This section introduced and defined the Retrieval and Analysis process. The next section describes the components used to implement this process and their integration into the VIZZANALYZER. It also covers the components used to realize the remaining goals of the thesis.

4.2 Components and Dataflow

Figure 4.2 gives an overview of all software components and data storage systems involved in the implemented solution of the thesis as well as the dataflow between them. The grey-colored rectangles represent preexisting or third-party components, whereas the white-colored rectangles represent newly implemented components and sub-components. The solid arrows stand for dataflow.

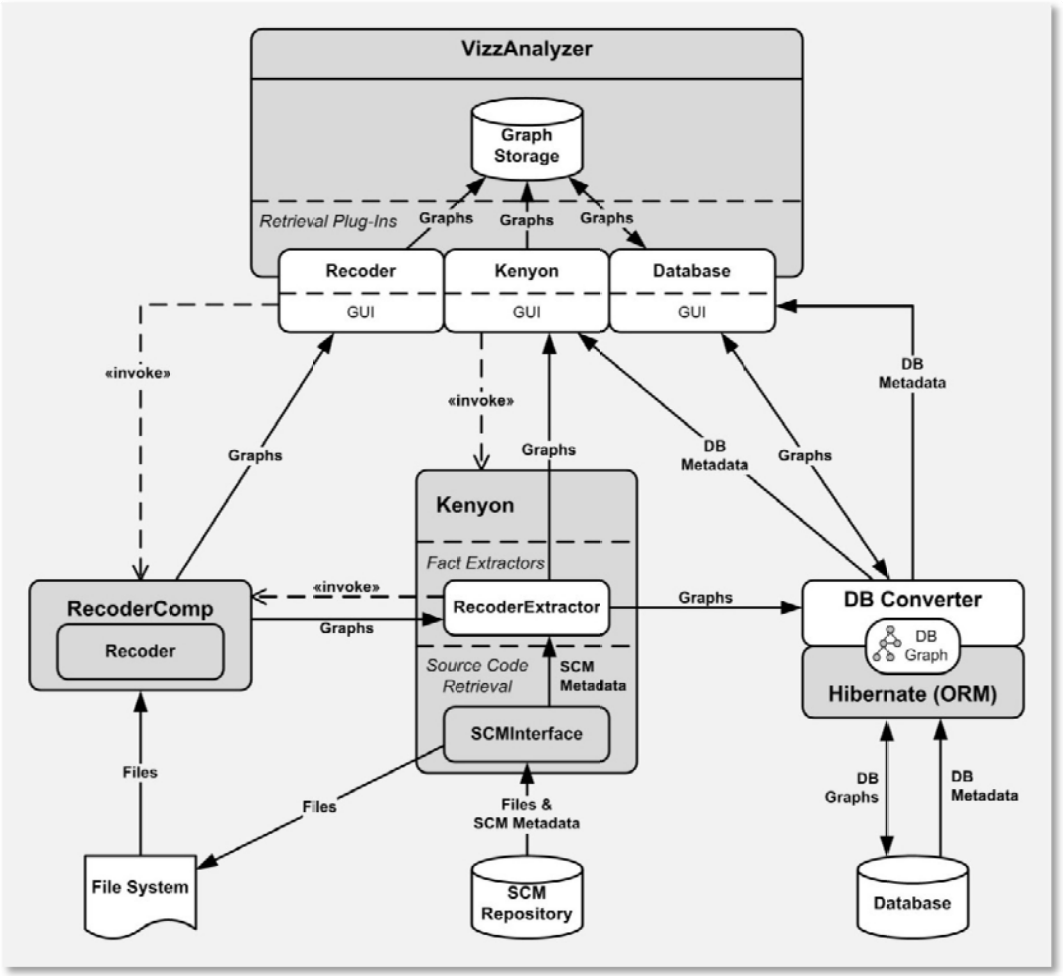

Figure 4.2: Dataflow between the integrated components and data storage systems

Data Storage Systems

The four data storage systems Graph Storage, File System, SCM Repository, and Database shown in the figure have already been introduced in the previous section. The internal graph storage of VIZZANALYZER is special in the respect that its data is only persistent during runtime of VIZZANALYZER. Therefore, the graphs it contains are runtime instances as well.

VizzAnalyzer and Plug-Ins

The VIZZANALYZER itself is the main application into which other components can be integrated via its plug-in interface. The figure only shows the interface for retrieval plug-ins. The two other interfaces for analysis and visualization plug-ins are neglected, as they are not relevant for the thesis. Besides the preexisting Recoder plug-in, the two new plug-ins KENYON and Database have been implemented. All three plug-ins possess a GUI for user interaction, which are based on a common GUI framework, described in Section 4.2.2 below. The existing Recoder plug-in has been reengineered in order to accommodate to this GUI.

Recoder Plug-In

The Recoder plug-in allows the user to configure and run the RECODER tool, which is part of the RECODERCOMP package and responsible for transforming source code into graphs compatible with the internal graph storage of the VIZZANALYZER. The plug-in itself is not relevant for the thesis and is mentioned here only for the sake of completeness. After RECODERCOMP has been invoked by the plug-in, it analyzes source code stored on the local file system, running without further user interaction. When the analysis is completed, the Recoder plug-in retrieves the generated graphs from RECODERCOMP and passes them on to the graph storage.

Database Plug-In

The Database plug-in offers access to a database management system (DBMS). The user can load graphs from the database (Use Case UC2), manage the graphs inside the database (Use Case UC3), and store graphs into the database (Use Case UC4). Separate GUIs have been implemented for each of these tasks. The plug-in receives database metadata from the DB Converter component; e.g., the lists of all graphs and projects stored inside the database. When graphs should be loaded from or stored into the database, they are received from the internal graph storage of VIZZANALYZER and passed on to the DB Converter or vice versa.

DB Converter and Hibernate

The DB Converter component is responsible for all communication with the database. However, it has no direct connection to the database but uses the object-relational mapping (ORM) tool HIBERNATE instead, which maps the graph data structure to entities of a relational database. Thereby, HIBERNATE offers access to a relational DBMS in an object-oriented way. Besides its role as an interface to HIBERNATE, another task of the DB Converter is to convert graphs objects into a different graph data structure, called DBGraph data structure, which is compatible with the ORM system of HIBERNATE. This is necessary, as it is not possible to create an object-relational mapping for the rather sophisticated graph data structure used by VIZZANALYZER without changing it. Changes to the graph data structure might have led to incompatibilities and therefore, each graph object has to be converted into a DBGraph object before it can be written to the database. The DB Converter also works in the opposite direction, restoring the original graph object from a DBGraph object.

Kenyon Plug-In

The Kenyon plug-in serves as a wrapper for the KENYON tool, used to implement the Retrieval and Analysis Process described in the section above. KENYON itself does not provide a GUI and therefore the plug-in is providing one for the user in order to configure and run KENYON. The plug-in receives database metadata from the DB Converter component, which are necessary for the GUI, e.g., available projects or status information of the database. When KENYON is invoked by the plug-in, it first initializes the SCMInterface with metadata from the SCM repository in order to determine which versions have to be retrieved. It then starts to retrieve the source code files of the first version, storing them on the local file system (Source Code Retrieval). After that, KENYON sequentially invokes the configured Fact Extractors and Metric Loaders to process the retrieved source code. One Fact Extractor, the RecoderExtractor, has been implemented for the thesis, whereas Metric Loaders have not been necessary. When the RecoderExtractor has finished, KENYON retrieves the next version and calls the RecoderExtractor again. This procedure continues until the last version has been retrieved and processed.

RecoderExtractor

The RecoderExtractor realizes the remaining steps of the Retrieval and Analysis Process. It first invokes RECODERCOMP in order to analyze the retrieved source code, generating one or more graphs (Analysis). The graphs are then annotated with evolutional information which the RecoderExtractor receives as SCM metadata from the SCMInterface (Graph Annotation). Finally, the graphs are passed to the internal graph storage of VIZZANALYZER by passing them through the Kenyon plug-in and/or they are written to the database, using the DB Converter (Graph Storage).

Summary

This section introduced the two new plug-ins Kenyon and Database that have to be implemented for the thesis as well as the fact that they comprise a common GUI framework. The next two sections will cover the plug-in interface of VIZZANALYZER and the GUI framework before the plug-ins themselves are described in the following.

Figure 4.3 shows a package diagram of the implemented components. It illustrates the architecture on a high level. However, its main purpose is to list all implemented class.

It is noteworthy that the implemented solution does not use the database storage capabilities of KENYON (cf. Section 2.3) in order to write graphs to database, even though it uses a similar approach. One reason for this decision is that it was unclear whether the data structures used by KENYON for storing analysis results into a database are flexible enough to accommodate the graphs of VIZZANALYZER without loss of information. Another reason is that the DB Converter component must be independent from the Kenyon plug-in, as the user should be able to manually write graphs to database (Requirement R18).

The solution does not allow for converting and storing the generated graphs automatically into other graph formats. However, this functionality could easily be added, e.g., by utilizing KENYON's Metric Loader mechanism and the conversion adapters provided by VIZZANALYZER.

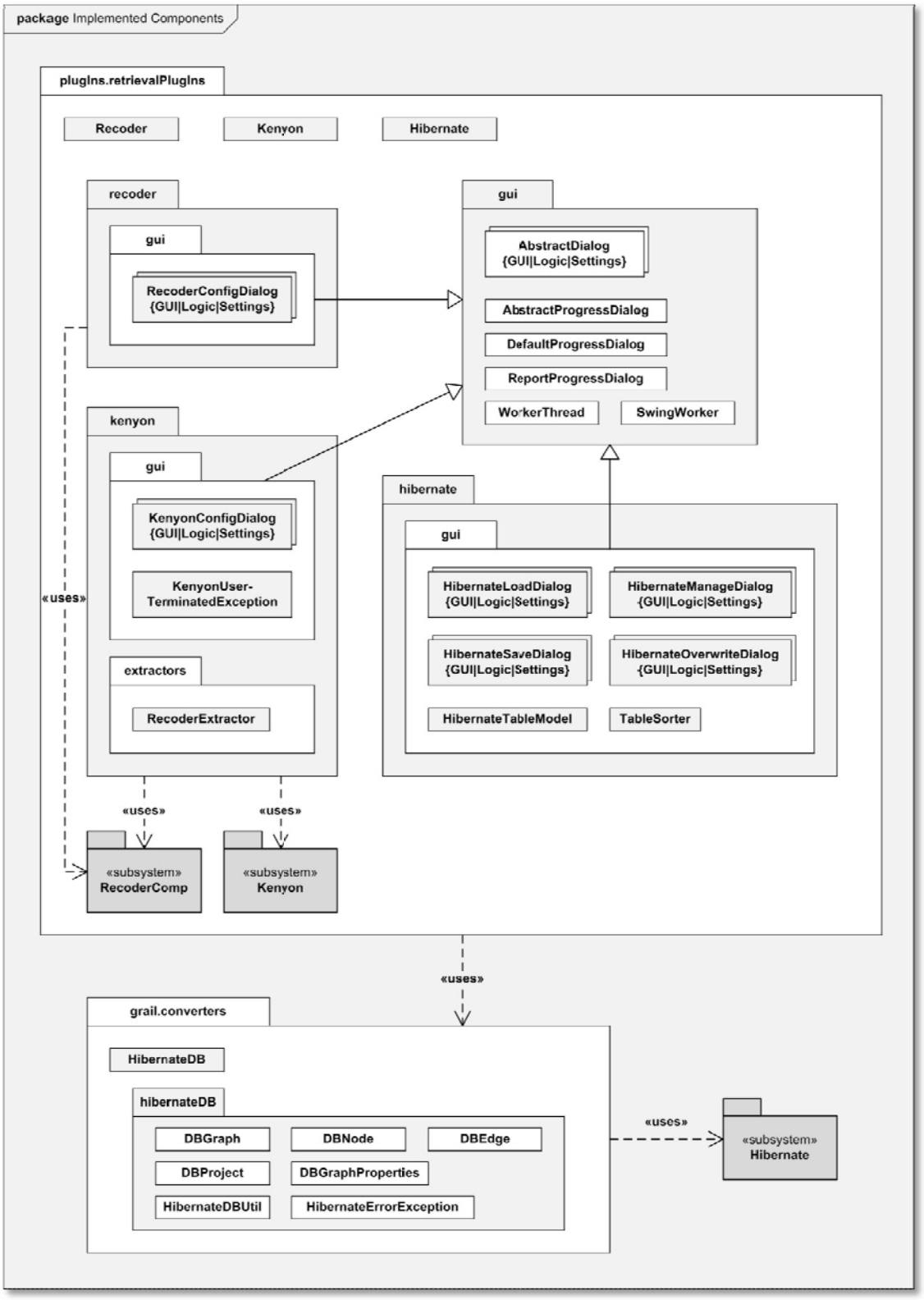

Figure 4.3: Package diagram of implemented components

4.2.1 Plug-In Interface

It has already been mentioned in Section 2.2.3 that the plug-in architecture of VIZZANALYZER contains three variation points for analysis (a), information extraction (b), and visualization (c) plug-ins, which are organized as directories containing the

wrapper classes for the respective plug-ins. Technically, the three variation points are all realized in the same way.

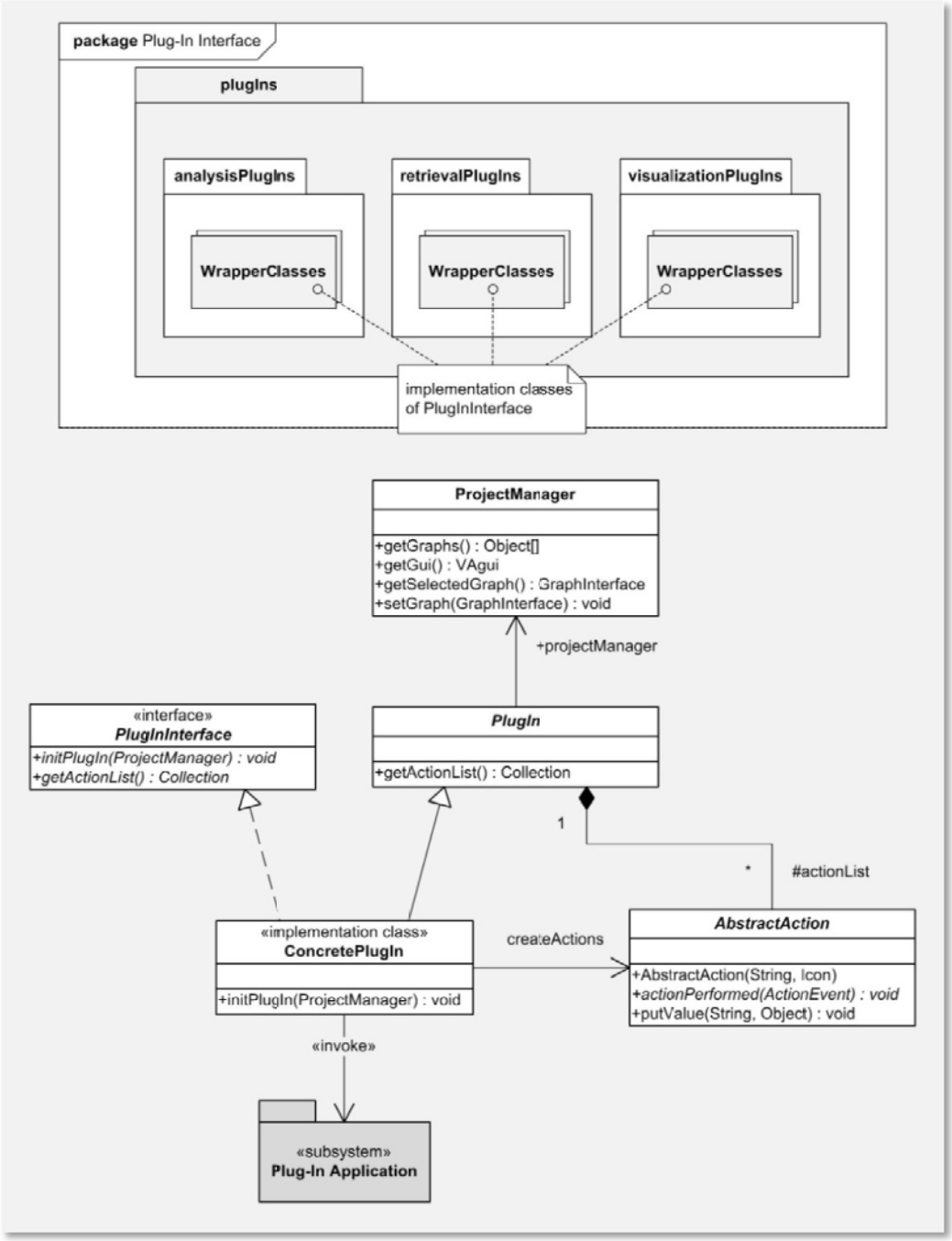

Figure 4.4: Package and class diagram of the plug-in interface

Figure 4.4 illustrates the concept by showing a package and class diagram. The directories containing the wrapper classes are represented as the sub-packages of the package `plugIns`, named `analysisPlugIns` (a), `retrievalPlugIns` (b), and `visualizationPlugIns` (c). The wrapper classes inside these packages have to implement the interface `PlugInInterface`, defining the methods `initPlugIn()` and `getActionList()`.

When VIZZANALYZER starts, it creates instances of all wrapper classes and calls their `initPlugIn()` methods in order to initialize the plug-ins. By calling this method, the reference of the `ProjectManager` object is passed to the plug-in, providing it with an interface for the internal graph storage of VIZZANALYZER as well as its GUI. This

allows the plug-ins to read and write graphs and to access the GUI of VIZZANALYZER, e.g., to attach new child dialog windows to it.

During the initialization process, VIZZANALYZER creates new menu entries in its main GUI for the available plug-ins by calling their `getActionList()` methods. Each wrapper class has to create a number of `AbstractAction` objects which this method returns. The objects contain information for the menu entries created by VIZZANALYZER, such as its name, a short description, or an icon, which are passed through their constructors and the `putValue()` method. The `AbstractAction` objects also implement the `actionPerformed()` method which is called when the user selects the menu entry of the corresponding action. The code in this method usually invokes the actual plug-in application.

The diagram also shows the class `PlugIn` which offers a default implementation for some methods of `PlugInInterface`. The wrapper classes, exemplary represented in the diagram as `ConcretePlugIn`, can extend this abstract class.

4.2.2 Common GUI framework

One goal of the thesis is to offer the user a graphical user interface (Feature F07). It follows from Use Case UC1 that a dialog window for the retrieval and analysis process is necessary. The Use Cases UC2, UC3 and UC4 require even more dialog windows for loading and managing graphs stored inside a database as well as saving graphs to a database. Furthermore, a progress window should be displayed during time-intensive operations (Requirement R20).

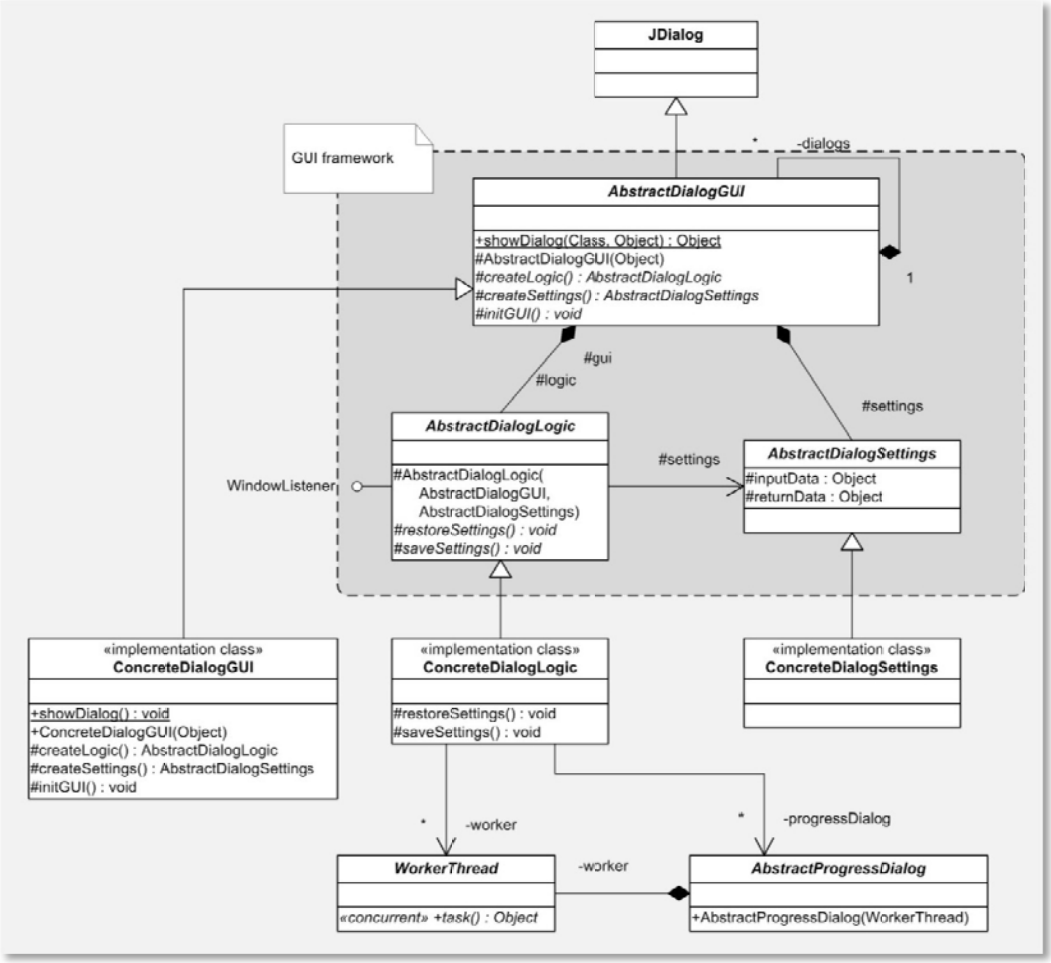

Figure 4.5: Class diagram of the common GUI framework

Instead of customizing new classes from scratch for each of the required dialogs, a common GUI framework was designed which can be reused by the implementation classes of the dialog windows. Figure 4.5 shows a class diagram of the classes involved in this framework.

Framework Classes

The actual framework consists of the three abstract classes `AbstractDialogGUI`, `AbstractDialogLogic`, and `AbstractDialogSettings`. They provide the core functionality and have to be implemented by each dialog window, exemplary represented in the diagram as `ConcreteDialogGUI`, `ConcreteDialogLogic`, and `ConcreteDialogSettings`. In the following, these classes will sometimes be referred to as GUI, Logic and Settings respectively. The main idea behind this partitioning is to isolate the code responsible for displaying the dialog and its elements from the code responsible for handling user interaction. Also, it should be possible to maintain the state of the dialog in an extra object, which can be accessed by other components that should not have direct access to the GUI. The three abstracted classes of the framework are described below.

`AbstractDialogGUI` defines how the dialog window is displayed to the user. It contains and initializes all graphical elements of the dialog, such as tables, text, or buttons and also configures the layout of these elements inside the displayed window. It extends the `JDialog` class of the JAVA SWING framework, providing the basic functionality of a dialog window in JAVA. `AbstractDialogGUI` also serves as the *façade* of the framework for calling classes. This means that a calling class does not need to interact with other classes of the GUI framework. Instead, it just has to call the class method `showDialog()` of the `ConcreteDialogGUI` that it wants to be displayed, which in turn calls the `showDialog()` method of `AbstractDialogGUI` together with a `Class`-parameter of its own type. The creation of the `ConcreteDialogGUI` instance is then handled automatically using reflection techniques. Once instantiated, `AbstractDialogGUI` keeps a reference of the concrete dialog in the class attribute `dialogs` until the system exists, even if the dialog window is closed. Optionally, the `showDialog()` method can be used to pass input or return parameters to the GUI and the calling class respectively. The GUI elements of the dialog have to be initialized by the implementation of the `initGUI()` method. The methods `createLogic()` and `createSettings()` are factory methods for creating instances of concrete Logic and Settings objects whenever a new instance of `ConcreteDialogGUI` is created.

`AbstractDialogLogic` handles the user interaction with the elements of the dialog and thereby isolates the logic of the GUI from its elements. It provides a default implementation for the `WindowListener` interface of the JAVA Abstract Window Toolkit (AWT) for handling window events. The `ConcreteDialogLogic` class has to implement other listener interfaces (e.g., `ActionListener` or `MouseListener`) as necessary in order to observe other user interactions with the dialog window. It also has to implement the methods `restoreSettings()` and `saveSettings()` which are responsible for saving the state of the dialog to the `ConcreteDialogSettings` object, e.g., when the dialog window closes, or to restore it, e.g., when the dialog is opened again. In order to fulfill its tasks, it is necessary for the dialog's Logic to have references of both the GUI and the Settings objects. Hence, they are passed as constructor arguments.

`AbstractDialogSettings` is a simple data structure for maintaining the state of an open dialog window. It stores all information necessary to fully restore the elements of the GUI and also provides default values for the initialization of them. However, it is not capable of saving or restoring the settings by itself, as it has no references of the GUI object. Instead, the Logic is responsible for these tasks. Another reason for the Settings class is to give other components access to the configurations made by the user

without granting them access to other parts of the GUI. The implementation of `AbstractDialogSettings` only provides the capability to store the input and return parameters of the GUI. All other attributes have to be defined by the implementation of `ConcreteDialogSettings`.

WorkerThread Class

Two other classes not directly part of the common GUI framework but still important for it are `WorkerThread` and `AbstractProgressDialog`. The dialog's Logic can encapsulate the code of time-intensive operations and run them concurrently in a separate thread in order to protect the GUI from being blocked by these operations. The `WorkerThread` is responsible for creating and managing these threads. The dialog's Logic creates an instance of `WorkerThread` for each operation that should run in a separate thread, implementing the execution code of the operation in the `task()` method.

AbstractProgressDialog Class

The `AbstractProgressDialog` class provides the core functionality for displaying a dialog window that monitors a concurrently running operation encapsulated in a `WorkerThread` object. The default implementation of this abstract class shows a progress bar and status messages. The user is able to stop the operation by clicking on a cancel button.

The concept behind the `WorkerThread` and `AbstractProgressDialog` classes are explained in greater detail in Chapter 5, as it is a quite complex topic by itself.

4.3 Kenyon Plug-In

In order to realize the Retrieval and Analysis Process described in Section 4.1, a plug-in has been implemented for VIZZANALYZER, called Kenyon plug-in. It follows VIZZANALYZER's plug-in architecture and implements the common GUI framework, which both have been described in the section above. This section first covers the architecture of the plug-in, before taking a closer look at the implemented GUI dialogs.

4.3.1 Architecture

The architecture of the Kenyon Plug-in consists of a wrapper class, GUI classes, KENYON itself and the class `RecoderExtractor`. They are explained in this section.

Wrapper Class

The architecture of the Kenyon plug-in is illustrated in Figure 4.6 as a class diagram. It shows the wrapper class `Kenyon`, linking the plug-in to the VIZZANALYZER by implementing the `PlugInInterface`. The plug-in is initialized when VIZZANALYZER calls the `initPlugIn()` method at its own start-up, passing a reference of the `ProjectManager` object as an argument. The class attributes, necessary for the `RecoderExtractor`, are set by the dialog's Logic object and the wrapper class itself. Only one action is defined during the initialization and displayed in VIZZANALYZER's menu: It allows the user to invoke the main dialog window of the Kenyon plug-in.

GUI Classes

The dialog window is represented by the `KenyonConfigDialogGUI`, `KenyonConfigDialogLogic`, and `KenyonConfigDialogSettings` classes, which are implementation classes of the common GUI framework. The dialog opens when the user selects Kenyon from VIZZANALYZER's Frontends menu, causing the `Kenyon` wrapper class to call `showDialog()` of `KenyonConfigDialogGUI`.

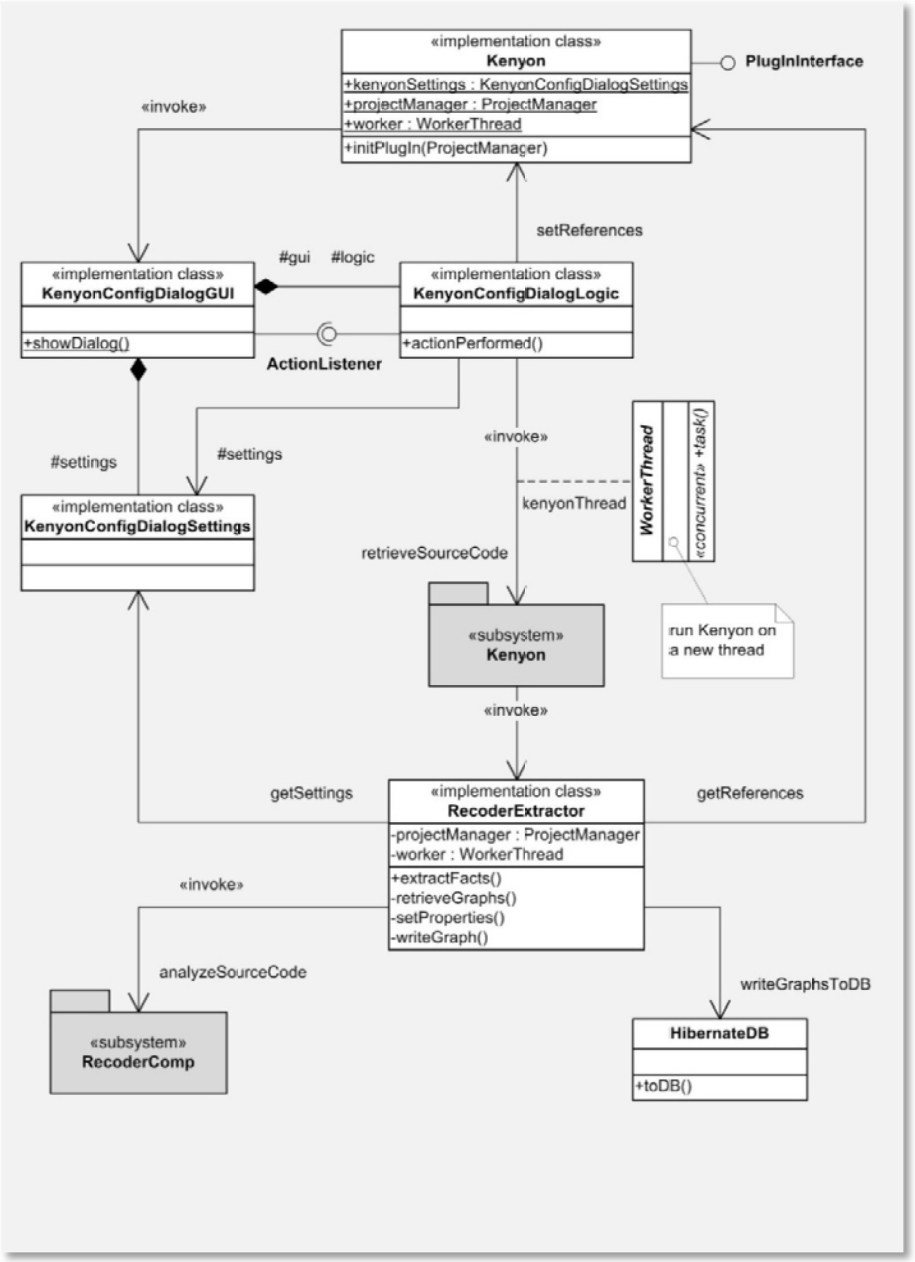

Figure 4.6: Class diagram of the Kenyon plug-in

The Logic of the dialog implements the `actionPerformed()` method of the `Action-Listener` AWT interface in order to observe the GUI and react to user interaction. When the user selects to run Kenyon, it stores the state of the dialog into the Settings object and sets a reference of it inside the wrapper class. It also sets a reference of the `WorkerThread` object used to start KENYON in a new thread. The references set inside the wrapper class are necessary for the execution of `RecoderExtractor` later. KENYON itself offers no possibility for a calling class to pass input parameters to its Fact Extractors and hence the only way for extractors to receive information from "outside" KENYON is to set class attributes. With the requirement of extendibility in mind, the class attributes have been placed inside the `Kenyon` wrapper class, to be accessible by future extractors as well.

Kenyon

After KENYON has been invoked, its task is to access the SCM repository and retrieve the source code of different versions (Requirement R11) according to the settings made

in the configuration file. As already mentioned, it is doing this in a sequential way: first retrieving the source code of the current version and then calling the Fact Extractors for further processing until all versions have been retrieved, analyzed, and stored.

RecoderExtractor Class

The only Fact Extractor currently implemented for the plug-in is the `RecoderExtractor` which is instantiated by KENYON. Its `extractFacts()` method is called by KENYON, each time a version has been retrieved from the SCM repository. The method then calls the RECODER tool (integrated into VIZZANALYZER as the RECODERCOMP package) to analyze the retrieved source code, resulting in a number of graphs (Requirement R12). The `retrieveGraphs()` method retrieves the generated graphs from RECODER and annotates them with evolutional information, provided by KENYON from the SCM repository, using the `setProperties()` method (Requirement R13). The method `writeGraph()` provides the functionality to store the graphs into database (Requirement R14) or into the internal graph storage of VIZZANALYZER as specified by the user. The `RecoderExtracor` receives a reference of the Settings object from the `Kenyon` wrapper class, set earlier by the dialog's Logic. It also receives references of the `ProjectManager` and `WorkerThread` objects for writing graphs to VIZZANALYZER's graph storage and setting status messages for the progress dialog respectively.

4.3.2 Dialog Windows

This section describes the dialog windows of the Kenyon plug-in, consisting of a main dialog window and a progress window.

Main Dialog Window

A dialog window has been implemented to allow the user to configure all necessary settings for the process to run. The dialog consists of three tabs:

The Config Files tab (cf. Figure 4.7) allows the user to choose the configuration files for KENYON as well as RECODER (Requirement R01). The user can also set some additional options, such as a default action when RECODER fails to analyze one of the retrieved versions (Requirement R03) or the graph types to be generated by RECODER (Requirement R04).

The Properties tab (cf. Figure 4.8) contains options for the annotation of the retrieved graphs with evolutional information from the SCM repository. The user can select the properties that should be attached to the graphs (Requirement R05). The values can either be absolute or relative to the root of the repository and the first version can be treated differently in this respect (Requirement R07). It is also possible to specify the relevant graph and node types (Requirement R06).

The Database tab (cf. Figure 4.9) gives the user the option to write the generated graphs to a database. Alternatively, it is possible to write the graphs to the internal graph storage of VIZZANALYZER (memory), or to both (Requirement R08). If the user selected to write graphs to database, a project and a description for the graphs have to be specified

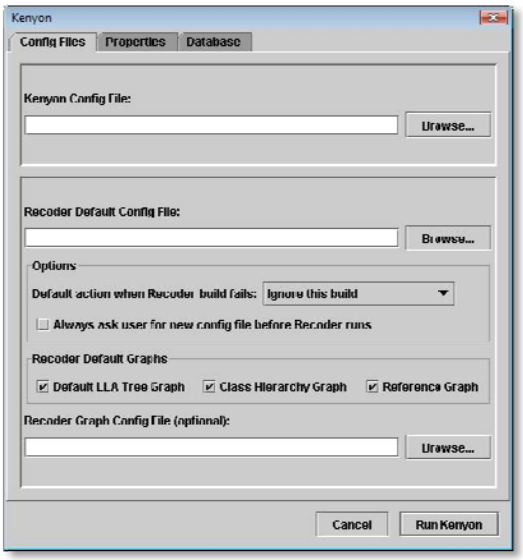

Figure 4.7: Main dialog window of the Kenyon plug-in (Config Files tab)

as well (Requirement R09). The description can be generated automatically (Requirement R10). Before the dialog window opens, the wrapper class probes the database for availability (Requirement R19). If this fails, the Database tab will only display a notification message.

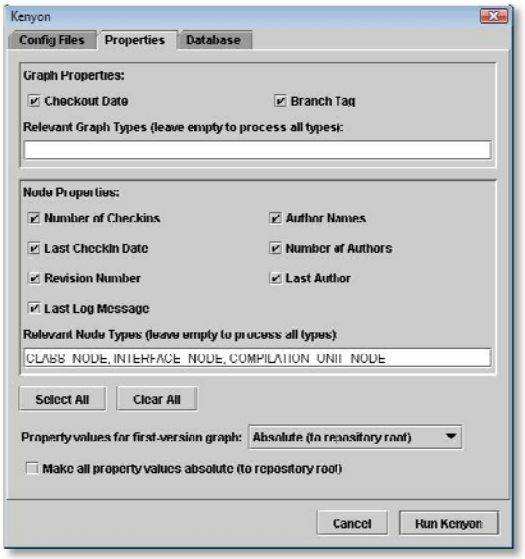

Figure 4.8: Main dialog window of the Kenyon plug-in (Properties tab)

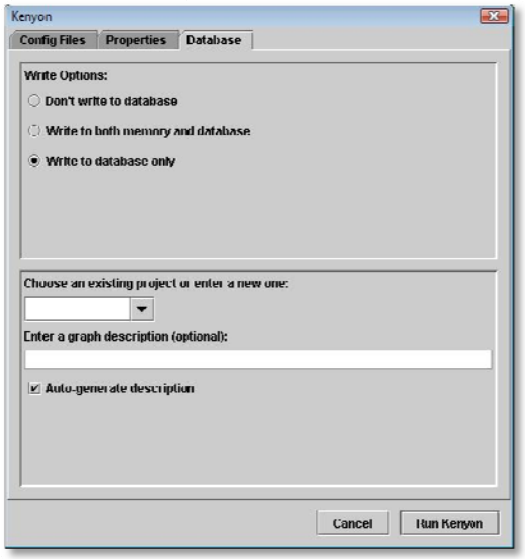

Figure 4.9: Main dialog window of the Kenyon plug-in (Database tab)

Progress Window

KENYON can be started by clicking on the Run Kenyon button after all necessary information has been filled in. The dialog validates the information first, displaying a notification window if data is missing or invalid (Requirement R02). The dialog window then closes, switching to a progress window (cf. Figure 4.10) that shows the current status of the process and a history of events (Requirement R20).

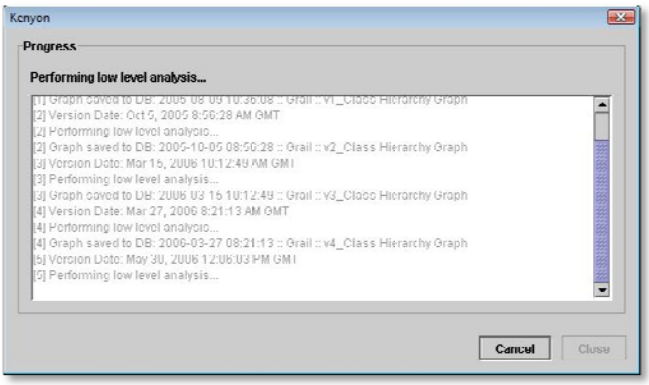

Figure 4.10: Progress window of the Kenyon plug-in

The user can stop the process by clicking on the Cancel button. When the process is completed (or has been canceled), the window stays open until the user clicks on the Close button, which then becomes available.

4.4 Database Plug-In

The Database plug-in has been implemented to give the user access to a database management system (DBMS). It provides the capability to load graphs from a database,

to manage the graphs inside a database, and to store graphs into a database. This section describes the architecture of the Database plug-in as well as the different dialog windows implemented for each of the tasks mentioned above.

The actual plug-in only consists of the wrapper and GUI classes. However, other database related components are also mentioned here. These components are responsible for establishing the connectivity to a DBMS. They can be seen as independent components that do not rely on the Database plug-in in order to work but rather are available for all plug-ins.

4.4.1 Architecture

The class diagram in Figure 4.11 gives an overview of the plug-in's architecture.

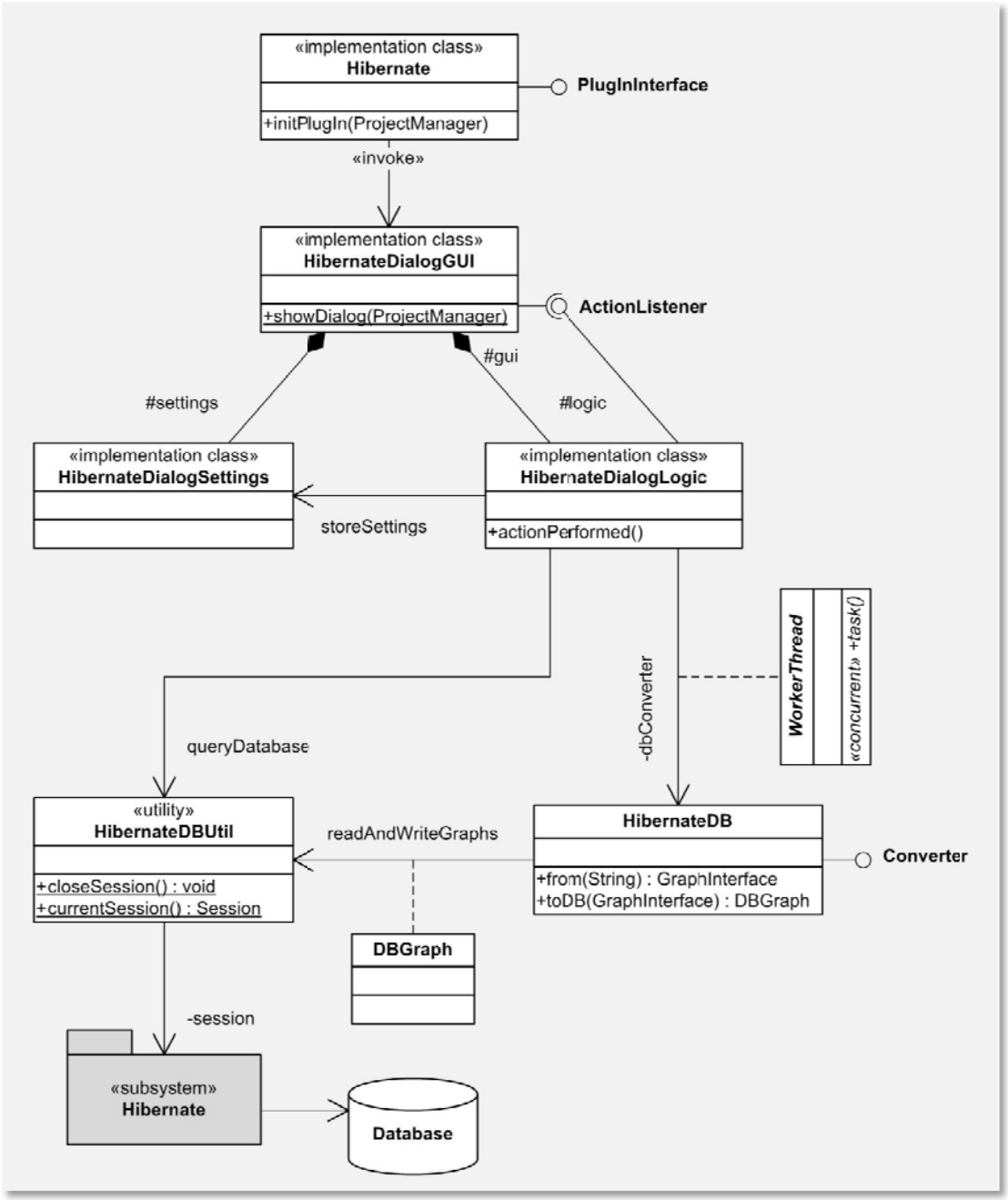

Figure 4.11: Class diagram of the Database plug-in

Wrapper Class

The wrapper class implementing the `PlugInInterface` is called `Hibernate`, analogous to the name of the ORM component used to connect to the DBMS. VIZZANALYZER passes a reference of the `ProjectManager` object to the wrapper class by calling the `initPlugIn()` method when it initializes the plug-in at system start-up. This method also defines the actions the plug-in is able to perform, displayed in the menu of the VIZZANALYZER window. These are: Load Graphs, Manage Graphs, Save Graphs, and Save Selected Graph. The plug-in uses the common GUI framework to create a dialog window for each of these actions. Save Graphs and Save Selected Graph use the same dialog window. An additional dialog window named Overwrite Graphs is needed for overwriting existing graphs.

GUI Classes

The class diagram does not show all implementation classes of the dialogs. Instead, it shows the classes `HibernateDialogGUI`, `HibernateDialogLogic`, and `HibernateDialogSettings` as representatives. The names of all implemented GUI classes are listed in Table 4.1.

Dialogs	Classes and showDialog Methods
Load Graphs	**`HibernateLoadDialog{GUI|Logic|Settings}`** `public void showDialog (ProjectManager pm)`
Manage Graphs	**`HibernateManageDialog{GUI|Logic|Settings}`** `public void showDialog ()`
Save Graphs and Save Selected Graph	**`HibernateSaveDialog{GUI|Logic|Settings}`** `public void showDialog (GraphInterface[] graphs)`
Overwrite Graphs	**`HibernateOverwriteDialog{GUI|Logic|Settings}`** `public void showDialog (String description)`

Table 4.1: GUI classes and showDialog methods of the Database plug-in

The user can open the dialogs by selecting an entry from the Database submenu of VIZZANALYZER's Frontends menu, causing the `Hibernate` wrapper class to call `showDialog()` of the corresponding GUI class. The argument of this method is different for each of the dialogs. The methods and their input parameters are listed in Table 4.1 as well.

The Load Graphs dialog takes a parameter of type `ProjectManager` (shown in the diagram), as it has to write graphs to the internal graph storage of VIZZANALYZER. The input parameter of the Save Graphs and Save Selected Graph dialogs is of type `GraphInterface[]` and holds references of the graphs that should be written to database. The Overwrite Graphs dialog has the existing graph's description of type `String` as input parameter. The Manage Graphs dialog does not require an input parameter.

The GUI classes of the Load Graphs and Manage Graphs dialogs are associated with two other classes not shown in the diagram: The `HibernateTableModel` and

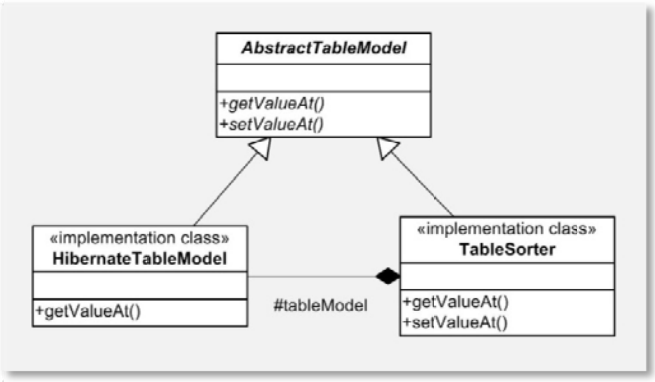

Figure 4.12: TableModel classes

TableSorter classes (the latter one was adapted from the Java Tutorials [30]). Both are subclasses of the SWING class AbstractTableModel, serving as a data model for SWING JTable components. They are illustrated in Figure 4.12. Class HibernateTableModel is the data model used by the table components of the dialogs named above for displaying the graph information stored in the database. It is associated with the TableSorter class which adds sorting functionality to a table model.

Dialogs	Concurrently Executed Operations
Load Graphs	• Load selected graphs from database
Manage Graphs	• Copy selected graphs between projects • Move selected graphs between projects • Delete selected graphs • Delete selected projects
Save Graphs and Save Selected Graph	• Write graph(s) to database
Overwrite Graphs	(no concurrently executed operations)

Table 4.2: Concurrently executed operations of the Database plug-in

The Logic of each dialog is responsible for observing its GUI and handling user interaction. They all implement the actionPerformed() method of the ActionListener AWT interface for this purpose. The Load Graphs and Manage Graphs dialogs additionally implement two other AWT and SWING interfaces: the MouseListener and the ListSelectionListener. The dialogs perform several time-intensive database operations which are encapsulated by the Logic within the task() method of WorkerThread objects and executed in separate threads. These concurrently executed operations are listed in Table 4.2. The Overwrite Graphs dialog only serves as a question dialog and therefore it has no operations that need to run concurrently.

The dialogs' Logic objects use the utility class HibernateDBUtil in order to query the database for information.

HibernateDB Class

When the Load Graphs, Save Graphs, and Save Selected Graph dialogs want to load or write graphs from or to database, they use the respective from() and toDB() methods of a HibernateDB object (Requirement R15). This class serves as a conversion adapter (cf. Section 2.2.3) between a DBMS and the VIZZANALYZER. Hence, it is located in the grail.converters package and implements the from() method of the Converter interface. The method takes a String object as argument, but unlike the other converters this string does not represent the contents of a graph file. Instead, it is the string representation of the Long identifier for a DBGraph object stored in the database.

The toDB() method works in the opposite direction, converting graphs to DBGraph objects first, before they are written to database. The method takes an argument of type GraphInterface.

DBGraph Class

As the data structure of VizzAnalyzer graphs cannot be mapped directly to database entities by the ORM system, they first need to be transformed to another data structure, the class DBGraph, before they can be written to database and vice versa. The mapping of the graph data structure classes to database entities is discussed in Section 5.2.2.

HibernateDBUtil Class

`HibernateDBUtil` is a utility class serving as an interface to the HIBERNATE ORM system which is responsible for all interaction with the DBMS. It provides a number of convenience class methods, e.g., a method that probes the database for availability (Requirement R19). Another task is to create and manage Hibernate sessions using the methods `currentSession()` and `closeSession()`. The main function of a `Session` is to offer create, read, and delete operations for instances of mapped entity classes.

4.4.2 Dialog Windows

This section presents and shortly describes the various dialog windows of the Database plug-in.

Load Graphs Dialog Window

The Load Graphs dialog window (Requirement R16), shown in Figure 4.13, consists of three panels and a button bar at the bottom of the window.

The Projects panel lists all projects names available in the database, including the number of graphs they contain. The user can select one or more projects at the same time. The corresponding graphs are then listed in the Graphs panel.

The Graphs panel contains a table that shows all graphs of the selected projects by listing the description, number of nodes and edges, and the creation and last-modified date of each graph in its own column. The user can select one or more graphs at the same time.

The Graph Details panel lists all graph properties, including their values, of the selected graph. If several graphs are selected, it shows the properties of the graph that has been selected last.

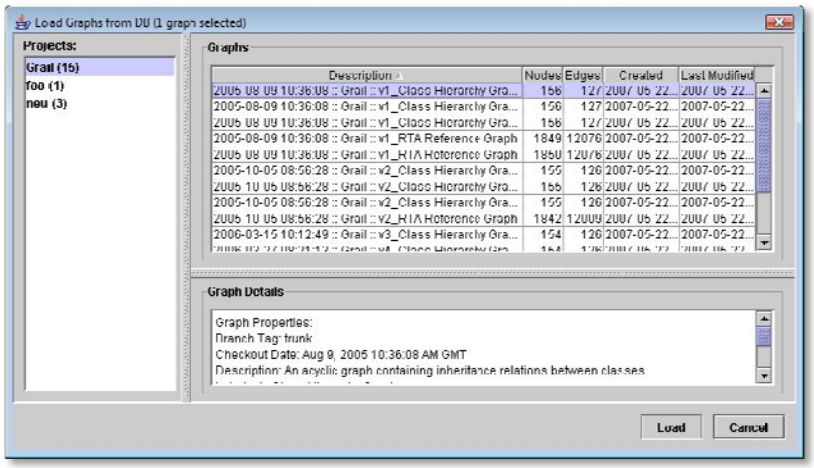

Figure 4.13: Load Graphs dialog window of the Database plug-in

The user can load the selected graphs by clicking the Load button. The dialog window then closes and a progress window is displayed instead.

Manage Graphs Dialog Window

The Manage Graphs dialog window (Requirement R17), shown in Figure 4.14, has the same layout and characteristics as the Load Graphs dialog, but also contains some additional buttons.

The Projects panel contains three buttons: the New button for creating a new project, the Rename button for renaming the selected project, and the Delete button for deleting the selected projects, including all graphs they contain. The New and Rename buttons

open a simple text input dialog when clicked. The Delete button display a confirmation dialog, switching to a progress window after the user has confirmed.

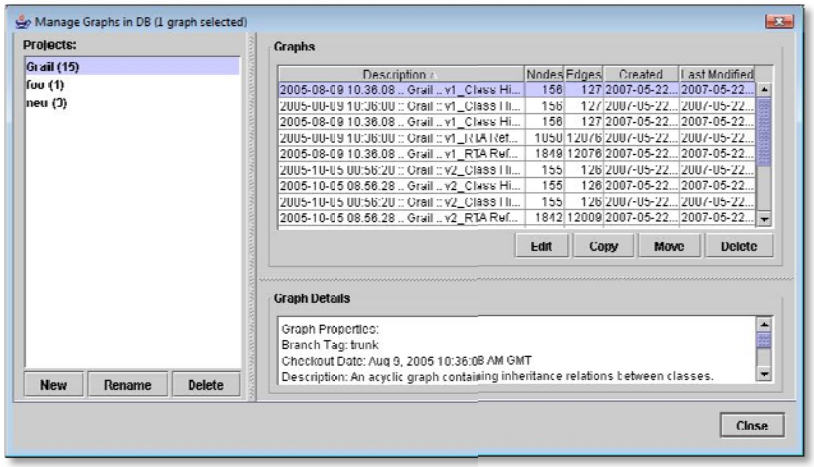

Figure 4.14: Manage Graphs dialog window of the Database plug-in

The Graphs panel contains four buttons: the Edit button for changing the selected graph's description, the Copy button for copying the selected graphs to a project, the Move button for moving the selected graphs to a project, and the Delete button for deleting the selected graphs. The Edit button opens a simple text input dialog when clicked. The Copy and Move buttons show a question dialog, asking for the destination project. They both switch to a progress window. The Delete button displays a confirmation dialog and also switches to a progress window after the user has confirmed.

Save Graphs and Save Selected Graph Dialog Window

The Save Graphs and Save Selected Graph dialog windows (Requirement R18) share the same layout and characteristics, except for the window's title. Figure 4.15 shows the Save Graphs window. The user can select the project to which the graphs should be written or enter the name of a new project. Optionally, a description for the graphs can be entered into the text field (Requirement R09). When the window opens, the text field contains the description of the first graph by default.

After the user clicked the Save button, the dialog closes and switches to a progress window. If one of the graphs written to the database has previously been loaded from the database and still exists there, the process pauses and the Overwrite Graph dialog window is displayed.

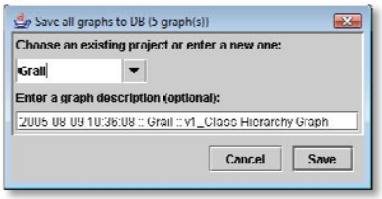

Figure 4.15: Save Graphs dialog window of the Database plug-in

Overwrite Graph Dialog Window

The Overwrite Graph dialog window is a question dialog, asking the user for an action when a graph written to database already exists there. It is shown in Figure 4.16. The user can choose to overwrite the existing graph, either keeping the existing graph description or providing a new one. Alternatively, it is possible to choose that the graph should be saved as a new graph or that the graph is skipped and not written to database.

When the "Apply to all graphs" box is checked, the dialog applies the user's decision to all remain-

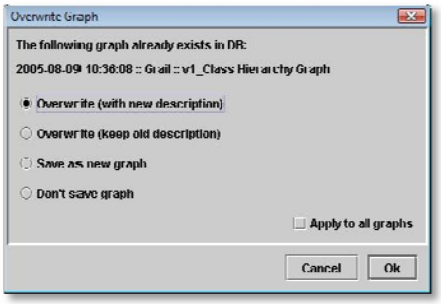

Figure 4.16: Overwrite Graph dialog window of the Database plug-in

ing graphs affected by the current operation, preventing the dialog window from appearing again until all graphs have been processed.

Progress Window

All dialogs of the Database plug-in use the same progress window to display progress and status of the current operation (Requirement R20). Figure 4.17 shows a progress window of the Load Graphs dialog as an example. The user can choose to stop the operation by clicking the Cancel button. The current operation is then notified, and when it has stopped the progress window closes.

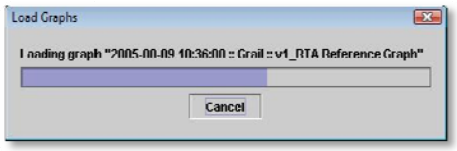

Figure 4.17: Progress window of the Database plug-in dialogs

4.5 Summary

This chapter gave an outline of the solution realized by the thesis. It defined the Retrieval and Analysis Process that needs to be implemented in order to fulfill the thesis' main goal. The components that were implemented or that are otherwise involved in the solution were explained as well as how they were integrated and how they interact.

The plug-in architecture of VIZZANALYZER and the common GUI framework were also introduced, as they provide the foundation for the implemented plug-ins. The architecture of the Kenyon and Database plug-ins were then explained on the basis of class diagrams. The GUI of each plug-in was pictured by displaying figures of its dialog windows. They enable the reader to get a better understanding of how the plug-ins can be used and what they are capable of.

The content of this chapter also proofs that all use cases and functional requirements, defined in Section 3.3 and 3.4 respectively, have been implemented successfully. Table 4.3 gives an overview of the components implementing the functional requirements.

Plug-In	Use Case	Component	Requirement
Kenyon	UC1	Main dialog	R01 – R10
		KENYON tool	R11
		RECODERCOMP package	R12
		`RecoderExtractor` class	R13, R14
		Progress window	R20
Database	UC2	`HibernateDB` class	R15
		Load Graphs dialog	R16
	UC3	Manage Graphs dialog	R17
	UC4	Save Graphs and Save Selected Graph dialogs	R09, R18
	UC1 – UC4	`HibernateDBUtil` class	R19
		Progress windows	R20

Table 4.3: Implementing components of the functional requirements

The non-functional requirements defined in Section 3.5 have also been addressed: The Performance requirement has been taken into account by using concurrently running

threads for time-intensive operations. Naturally, it is also influenced by the actual implementation of the components. Reusability is guaranteed by the common GUI framework and the `HibernateDB` conversion adapter. Extendibility is provided by the KENYON tool itself and its Fact Extractor concept. The Operational Requirements have been addressed by implementing all components with the JAVA programming language which has also been used for realizing the VIZZANALYZER framework. All utilized third-party components have been implemented with JAVA as well.

5 Implementation

This chapter elaborates in more detail on the implementation of the solution. UML class and sequence diagrams as well as code examples are utilized to concretize architecture and design of certain components introduced in Chapter 4. Noteworthy issues that arose during the development process are described together with their corresponding methods of resolution.

5.1 Graphical User Interface

The common GUI framework and the worker thread concept had been introduced in Section 4.2.2. Although they are not directly involved in solving the main problem of the thesis, the realization of the Retrieval and Analysis Process, they are yet of great importance for the implementation. Both concepts help to improve the usability of retrieval plug-ins for the VizzAnalyzer. They also offer a high degree of reusability. The following two sections take a closer look at these important concepts.

5.1.1 Common GUI Framework

In this section, the common GUI framework introduced in Chapter 4 will be explained in greater detail by showing two sequence diagrams. They illustrate the typical sequence of message calls for the invocation of a dialog window and subsequent user interaction with this window respectively.

Dialog Invocation Process

Opening dialog windows with the common GUI framework is easy. The GUI class of each dialog offers a class method named `showDialog()` for this purpose. A call of this method is enough to invoke and open the dialog window. The framework uses factory methods to create the necessary dialog objects. Hence, the caller never has to create the dialog objects by itself.

The sequence diagram of Figure 5.1 explains the interaction between the participating classes and objects during the dialog invocation process. These are:

- the `ConcreteDialogGUI` class serving as a façade for the calling object,
- the `AbstractDialogGUI` class serving as the controller during the invocation process,
- the `ConcreteDialogGUI` object reflecting the GUI of the dialog,
- the `ConcreteDialogLogic` object reflecting the Logic of the dialog, and
- the `ConcreteDialogSettings` object reflecting the Settings of the dialog.

The process starts with a call of the dialog GUI's `showDialog()` method which in turn calls the `showDialog()` method of `AbstractDialogGUI`. The first method takes the dialog's parent frame, the window's title and an optional input parameter as arguments. The latter one adds a `Class` object of itself to this list. Code 5.1 lists the `showDialog()` method of `HibernateOverwriteDialogGUI` as an example.

```
static public String showDialog(Component parent, String title, String graphDesc) {
    String result = (String) AbstractDialogGUI.showDialog(frameComp, title,
             HibernateOverwriteDialogGUI.class, graphDesc);
    return result;
}
```

Code 5.1: showDialog() method of HibernateOverwriteDialogGUI

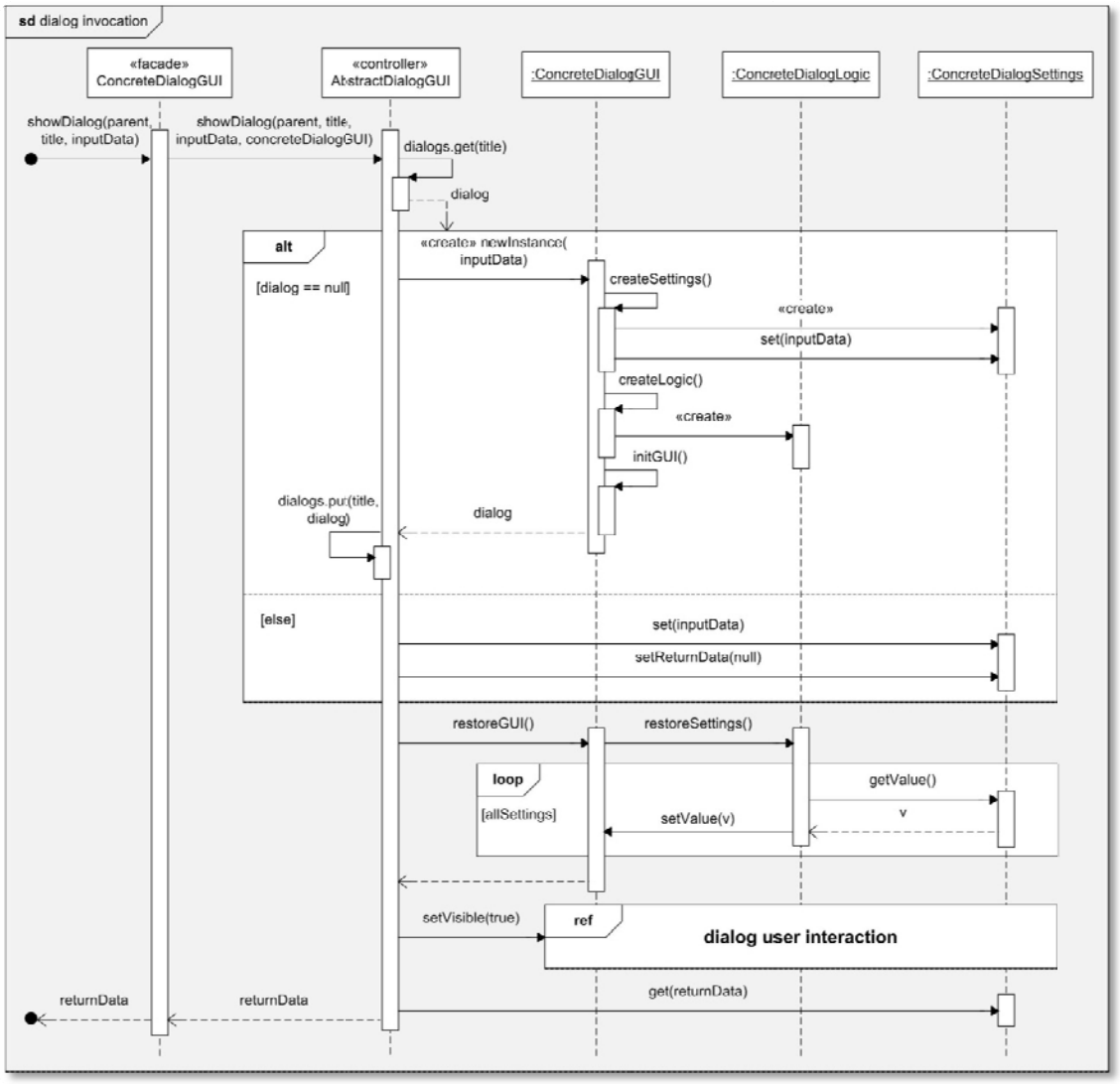

Figure 5.1: Sequence diagram of a dialog invocation process

The dialog in the example takes the graph description as an input parameter. The return parameter of `AbstractDialogGUI.showDialog()` is of type `Object` and therefore it needs to be casted to the correct return type of the dialog's `showDialog()` method. In this case, it is a `String` reflecting the user's choice.

`AbstractDialogGUI` holds references of all GUIs that have been previously created in a `Hashtable`, named `dialogs`, mapping the dialog titles to the according object references. Consequently, several instances of the same dialog can exist at the same time, e.g., for the Save Graphs and Save Selected Graph dialogs. However, the dialog's window title must be unique for each instance.

After `showDialog()` has been called, the method tries to get the dialog from the hashtable. If the returned object is `null`, the dialog has never been opened before and must be created first. This is done by using the `Class` object received as input parameter and reflection techniques.

The diagram also shows what happens during the construction of the dialog:

1. The constructor of the new GUI instance is called with the input parameter as an argument.

2. The constructor calls the factory method `createSettings()` in order to create a new instance of the Settings class. The method also stores the input parameter in the newly created Settings object.
3. The constructor calls the factory method `createLogic()` in order to create a new instance of the Logic class.
4. The constructor calls `initGUI()` in order to initialize the GUI of the dialog. The dialog window is then ready to be displayed.

The Settings object must be created before the Logic, as the Logic's constructor requires a reference of Settings as argument. An example of both factory methods is shown in Code 5.2. After the control flow has returned to `AbstractDialogGUI` and the new GUI object was created, its reference is put into the `dialogs` hashtable.

```
protected AbstractDialogLogic createLogic() {
     this.logic = new HibernateOverwriteDialogLogic(this, this.settings);
     return this.logic;
}

protected AbstractDialogSettings createSettings() {
     this.settings = new HibernateOverwriteDialogSettings();
     return this.settings;
}
```

Code 5.2: Factory methods of HibernateOverwriteDialogGUI

If the GUI has been returned successfully by the hashtable, the dialog was opened before. In that case, `showDialog()` simply stores the input parameter into the Settings of the dialog. The return value of the dialog, which is also stored inside the Settings object, is set to `null` as it has not been determined yet.

In the next step, `restoreGUI()` is called in order to restore the GUI elements to the state saved in the Settings object. For a newly created dialog, this means that default values are loaded. Even though a dialog that has already been created did not change its state since it was closed this step might be necessary; e.g., the Manage Graphs dialog must synchronize its lists of graphs and projects with the database.

The method delegates this task to the Logic by calling `restoreSettings()`. The implementation of this method is specific for each Logic class. The diagram shows an example of how the Logic restores the state of the GUI by loading values from the Settings object.

Finally, the dialog window is opened by calling the GUI's `setVisible()` method. The following user interaction is illustrated by another sequence diagram in Figure 5.2 below.

Eventually, when the window is closed, `AbstractDialogGUI` returns to the calling `showDialog()` method with the return value it got from the Settings object. At last, the value is returned to the caller.

Dialog User Interaction

Naturally, the user interaction is very specific to each instance of a dialog window. Therefore, the sequence diagram in Figure 5.2 can only give a general description of how user interaction with a dialog takes place. The use cases in Section 3.3 give a more detailed description of each case of user interaction with the implemented dialog windows.

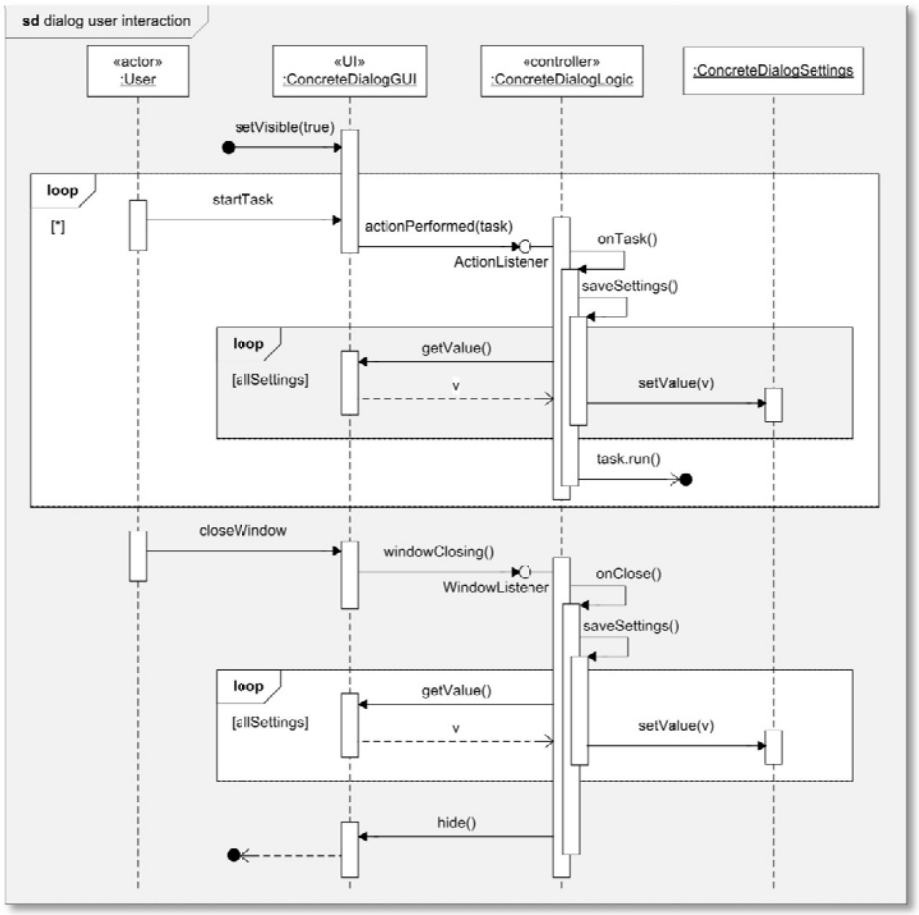

Figure 5.2: Sequence diagram of dialog user interaction

The participants in the diagram are:

- the `User` interacting with the dialog window,
- the `ConcreteDialogGUI` object reflecting the dialog's user interface,
- the `ConcreteDialogLogic` object serving as the controller during the user interaction, and
- the `ConcreteDialogSettings` object reflecting the Settings of the dialog.

The starting event in the diagram is the call of the GUI's `setVisible()` method, causing the window to be opened and displayed to the user. The interaction of the user with the dialog window can be described as a number of tasks, illustrated by the `startTask` message and the surrounding loop frame. A task can either be started actively by the user, e.g. when clicking on a button that triggers an operation, or implicitly; e.g., when the user selects a graph in the Load Graphs dialog the database has to be queried in order to display the graph's details.

However, the GUI itself is not responsible for handling the tasks. It rather delegates them to its associated Logic object, by registering the Logic with its GUI elements during initialization. Therefore, the Logic has to implement certain listener interfaces. A commonly used interface is the `ActionListener` AWT interface. It is able to listen to events of buttons and other GUI elements. In the diagram, the user starts tasks by interacting with just this kind of elements. The GUI, i.e., the effected element, then calls the `actionPerformed()` method of the interface together with an argument that identifies the action. The Logic then calls a corresponding method, named `onTask()` in the

diagram, to handle the action event. Code 5.3 shows the `actionPerformed()` method of `HibernateLoadDialogGUI` as an example.

```
public void actionPerformed(ActionEvent e) {
    String aCommand = e.getActionCommand();
    if (aCommand.equals("cancel")) {
        onCancel();
    } else if (aCommand.equals("load")) {
        onLoad();
    } else {
        System.err.println("WARNING: Unknown action command: " + aCommand);
    }
}
```

Code 5.3: actionPerformed() method of HibernateLoadDialogGUI

The subsequent actions are specific to each task. The Logic might first save the state of the dialog window by calling its own `saveSettings()` method before running the actual task. This is illustrated exemplary in the diagram by the `getValue()` and `setValue()` messages surrounded by a loop frame. The Logic then starts the actual task, represented by a lost message named `task.run()`, as it is not further elaborated.

The diagram shows another action, named `closeWindow`, caused by the user When a dialog window closes, the `windowClosing()` method of the `WindowListener` interface, which has to be implemented by all Logic classes, is called. The `AbstractDialogLogic` class offers a default implementation which is usually overwritten by implementation classes to call their own `onClose()` method instead. In the diagram, this is followed by a call of `saveSettings()` and finally a call of the GUI's `hide()` method. In fact, this is the exact same behavior of the default implementation.

The `hide()` method causes the window to be closed and returns the flow of control to the caller, represented by another lost message ending the sequence diagram.

5.1.2 Concurrency

This section describes details of the `WorkerThread` class and its associated `AbstractProgressDialog` class introduced in Section 4.2.2. Both are utilized by concrete Logic classes of the common GUI framework. The first one is used to run the execution code of time-intensive operations on separate threads. The latter one is used to display a progress window during the execution of these operations. Figure 5.3 shows a diagram illustrating the two classes as well as other associated classes.

Using concurrency is a necessity to prevent a GUI realized with SWING components from "freezing" while time-intensive operations are running [17]. This can happen, because most SWING methods are not thread safe and must execute on the event dispatch thread where all event-handling code is executed. If a task on the event dispatch thread does not finish quickly, unhandled events back up and the user interface becomes unresponsive. Therefore, time-intensive tasks should be executed on separate worker threads.

WorkerThread Class

Each task running on a worker thread is represented by an instance of the `WorkerThread` class. In order to create objects of this abstract class, subclasses have to be defined, e.g., by using anonymous inner classes (cf. Code 5.4).

`WorkerThread` provides a number of communication and control features:

- The `doInteraction()` method can be called when the worker thread has to run code on the event dispatch thread, e.g., to display a question dialog. The worker

thread stops and waits until the code, provided as an argument of type `Runnable`, was executed.

- The `onFinished()` method is automatically invoked on the event dispatch thread when the worker thread is finished. The default implementation of this method is empty and should be overwritten by subclasses.
- The `stop()` method can be called to cancel the worker thread. However, the thread will not exit immediately when this method is called but rather sets a flag that has to be checked in short intervals by the execution code of the thread. This ensures that the thread can exit gracefully.

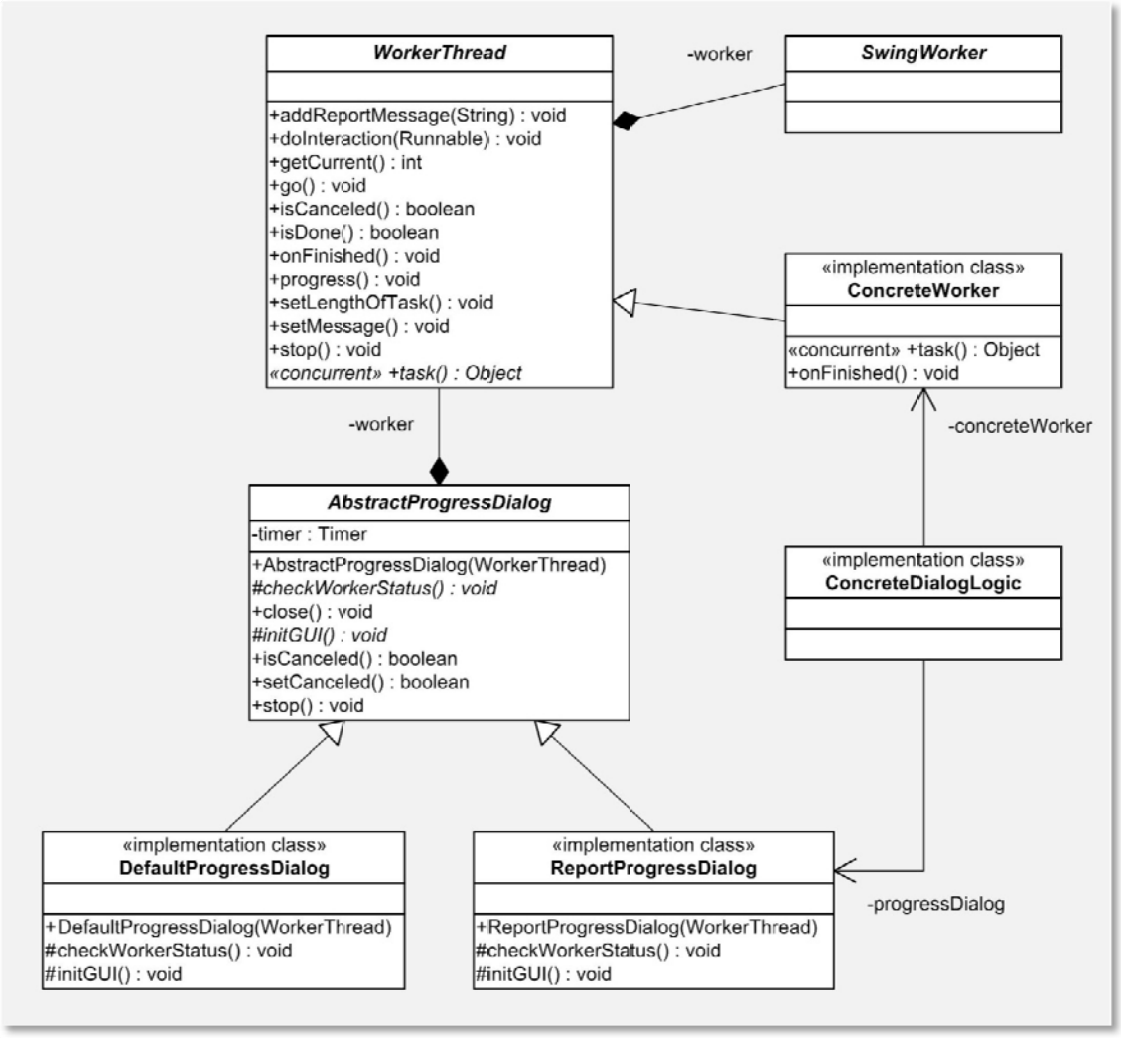

Figure 5.3: Class diagram of concurrency classes

The execution code of a worker thread has to be implemented by the `task()` method of its subclass. It is started when the `go()` method of `WorkerThread` is called, which in turn uses a `SwingWorker` object to create and run the actual thread. In this sense, `WorkerThread` could be understood as a wrapper class of `SwingWorker`. The class `SwingWorker` has been adapted from the Java Tutorials [30] and will not be further elaborated. Note that a different class with the same name has been added to the JDK in Java SE6.

The remaining methods illustrated in the diagram are used to inform an observing progress window about the current status of the concurrent task:

- The `addReportMessage()` method adds a new message string to the report message string used by `ReportProgressDialog`.
- The `getCurrent()` method returns the current value of the progress counter. It is called by `DefaultProgressDialog` to update its progress bar.
- The `isDone()` method indicates that the worker thread is finished.
- The `progress()` method increases the progress counter by one.
- The `setLengthOfTask()` method defines the total length of the task, i.e., how often `progress()` has to be called until the task is complete. It takes an argument of type `int`.
- The `setMessage()` method sets the current status of the task. It takes an argument of type `String`.

Code 5.4 shows an example of how a `WorkerThread` subclass can be defined. The code is actually used by the Manage Graphs dialog for moving graphs to a project.

The code for moving the graphs is surrounded by a `try` block, catching database errors. The corresponding `catch` block handles the errors by displaying an error window to the user. The code for opening the window is encapsulated in a `Runnable` object and executed by the `doInteraction()` method. As mentioned before, this is necessary as SWING components are not thread safe and therefore, corresponding code has to be executed on the event dispatch thread.

```
WorkerThread graphMoveThread = new WorkerThread() {
    public Object task() {
        try {
            setMessage("Gathering information...");
            // ... (calculate length of task)
            setLengthOfTask(numOfSel);
            // ... (iterate thru the projects and move the selected graphs)
                setMessage("Moving graph \""+graph.getDescription()+"\"");
                // ... (move graph)
                progress();
                // ...
            return null;
            // end of try
        } catch (final HibernateErrorException hee) {
            Runnable code = new Runnable () {
                public void run() {
                    HibernateDBUtil.showErrorDialog(hee, gui);
            }};
            doInteraction(code);
            return null;
            // end of catch
        }
        // end of task()
    }
    public void onFinished() {
        // ... (update GUI)
    }
    // end of graphMoveThread
};
```

Code 5.4: Example of a WorkerThread instance

The class diagram also reveals that both getter and setter methods of `WorkerThread` are *public*. It is obvious that the getter methods are public to allow an observing progress window to read the status of the task. The setter methods, including `addReportMessage()`, `doInteraction()` and `progress()`, are intended to be called exclusively by the execution code of the concurrently running task. However, these methods had to be declared public in order to allow their access of objects called during the execution

of the task. As an example, the `RecoderExtractor` has to be able to call these methods for displaying question dialogs or updating status information (cf. Section4.3.1).

As a consequence, the implementer has to be careful not to call these methods from another thread outside the task, as the `WorkerThread` itself does not guarantee that such calls are thread-safe. However, since the methods are only responsible for updating the task's status information and do not affect its actual execution, this risk is negligible.

AbstractProgressDialog Class

The `AbstractProgressDialog` class provides the basis for monitoring the status of a task running on a worker thread. In contrast to the `ProgressMonitor` class of the SWING framework, this class is more flexible as subclasses provide their own custom GUI. The GUI should be implemented by the `initGUI()` method.

Objects of this class check their associated `WorkerThread` object periodically for status updates by running a timer. The timer causes the `checkWorkerStatus()` method to be called in predefined time intervals (the default is 500ms). The method has to be implemented by subclasses in order to check if the status of the worker thread has changed and to take action accordingly, e.g., to update the GUI or to close the progress window when the worker thread has finished.

The following convenience methods are intended for implementing subclasses:

- The `close()` method stops the timer and closes the window.
- The `isCanceled()` method returns the cancel status, i.e., whether `setCanceled()` has been called before.
- The `setCanceled()` method sets the cancel status to true, e.g. when the user has clicked on the *Cancel* button.
- The `stop()` method stops the timer but leaves the window open.

Two subclasses of `AbstractProgressDialog` have been implemented to realize the requirement for displaying progress information (Requirement R20). They are described in the following two subsections.

DefaultProgressDialog Class

The `DefaultProgressDialog` realizes the progress windows of the Database plug-in. It consists of only three GUI elements:

- The status message displays the current status of the task; e.g., "Loading graph myGraph".
- The progress bar displays the current progress of the task.
- The Cancel button informs the worker thread that the user wants to stop the task. When clicked, the status message changes to "Waiting to cancel…". The window closes eventually when the worker thread has stopped.

An example of how this progress window looks like is shown by Figure 4.17 in Section 4.4.2.

ReportProgressDialog Class

The ReportProgressDialog realizes the progress window of the KENYON plug-in. It consists of the following GUI elements:

- The status message displays the current status of the task; e.g., "Performing low level analysis…".

- The report message area lists all report messages that have been made by the task during its runtime in chronological order.
- The Cancel button informs the worker thread that the user wants to stop the task. When clicked, the status message changes to "Waiting to cancel…". The window stays open even if the worker thread has stopped.
- The Close button allows the user to close the progress window. It is disabled until the worker thread has stopped.

An example of how this progress window looks like is shown by Figure 4.10 in Section 4.3.2.

Code Example

Code 5.5 shows a short example of how worker thread and progress window instances can be created. The code is actually used by the Manage Graphs dialog for moving graphs to a project.

```
protected void onGraphMove() {
    // ... (ask user for the name of the target project)
    WorkerThread graphMoveThread = new WorkerThread() {
        // ... (define worker thread here)
    };
    graphMoveThread.go();
    DefaultProgressDialog pd;
    pd = new DefaultProgressDialog(this.gui, "Move Graphs", graphMoveThread, true);
    pd.show();
    // end of onGraphMove()
}
```

Code 5.5: Example of creating worker thread and progress window instances

The `onGraphMove()` method is invoked when the user clicks on the Move Graphs button. The new thread starts running when the `go()` method of the `WorkerThread` object is called. The progress window is displayed when its `show()` method is called.

The parameters of the progress window's constructor are: the parent GUI, the title of the window, the worker thread and a `boolean` indicating whether the window should be modal or not. A modal window does not return the flow of control to its caller until it has been closed.

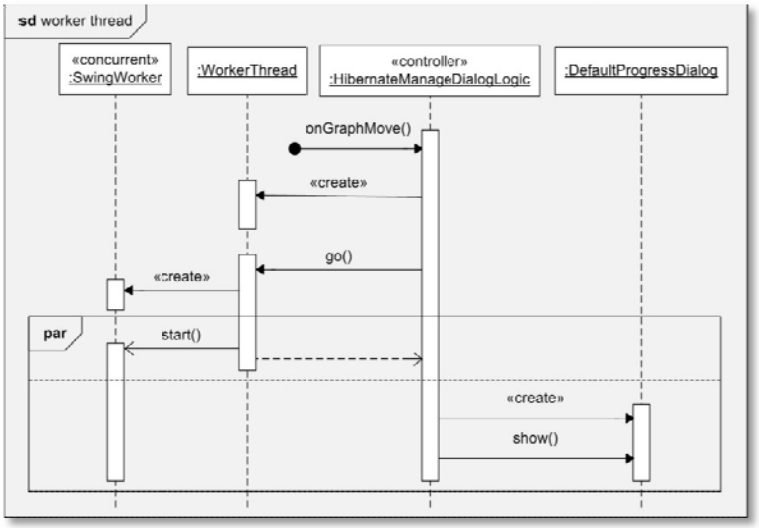

Figure 5.4: Sequence diagram of worker thread creation

The execution order for starting the worker thread and creating the progress window is not important; i.e., the `go()` method could just as well be called after creating the progress window. The only prerequisite is that the `WorkerThread` object has been realized before the progress window is created, since its reference has to be passed as an argument to the constructor of the progress window. If the progress window is modal, its `show()` method has to be called last (after `go()`). The sequence diagram in Figure 5.4 illustrates the code example.

5.2 Database Connectivity

The support of database management systems is one of the goals defined in Section 1.3. When the Database plug-in was described in Section 4.4, it had been mentioned that the connectivity with a DBMS is established by components which are not directly part of the plug-in. The core of these components is the object-relational mapping system HIBERNATE used by plug-ins to get access to database management systems. The basic functionality of HIBERNATE and its integration into the database components is therefore described in this section.

Another component which has barely been mentioned in Section 4.4.1 is the `DBGraph` class. It serves as an intermediate data structure between the graph data structure of VIZZANALYZER and the corresponding database entities. This section will take a closer look on the data structure and also discuss certain mapping issues.

5.2.1 Hibernate

HIBERNATE is an object-relational persistence and query service, making it possible to both store the state of a JAVA runtime object into a relational DBMS and to create objects from the corresponding entities [16]. This process is called object-relational mapping (ORM). It relieves the developer from manual coding with SQL queries and the JAVA DATABASE CONNECTIVITY (JDBC) API. Furthermore, HIBERNATE is compatible with all JDBC-compliant databases.

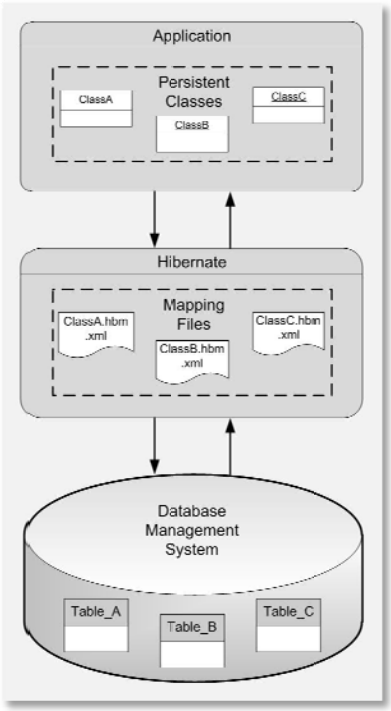

Figure 5.5: Object-relational-mapping with Hibernate

Basic Functionality

Figure 5.5 illustrates how HIBERNATE realizes its ORM functionality. The application has no direct access to the database but rather communicates with HIBERNATE instead. The classes mapped to the database are called persistent classes. Each persistent class has a mapping file with the same name as its corresponding class, plus the extension `".hbm.xml"`. The files map JAVA classes to database tables and JAVA data types to SQL data types using XML notation.

HIBERNATE can be configured by using a simple `hibernate.properties` file, a more sophisticated `hibernate.cfg.xml` file or complete programmatic setup. The configuration provides HIBERNATE with the name of the JDBC driver class, the location of the database, the user name, the password and the SQL dialect. Moreover, it specifies the mapping files for the persistent classes and other HIBERNATE specific options.

The implemented database component uses a XML configuration file and version 3.0.5 of HIBERNATE. During the component's development, a MySQL [31] database management system of version 4.1.10 and later 5.0.27 has been used, running on a Windows XP Professional SP2 and Windows Vista Business system respectively.

Architecture

The architecture of HIBERNATE is shown in Figure 5.6.

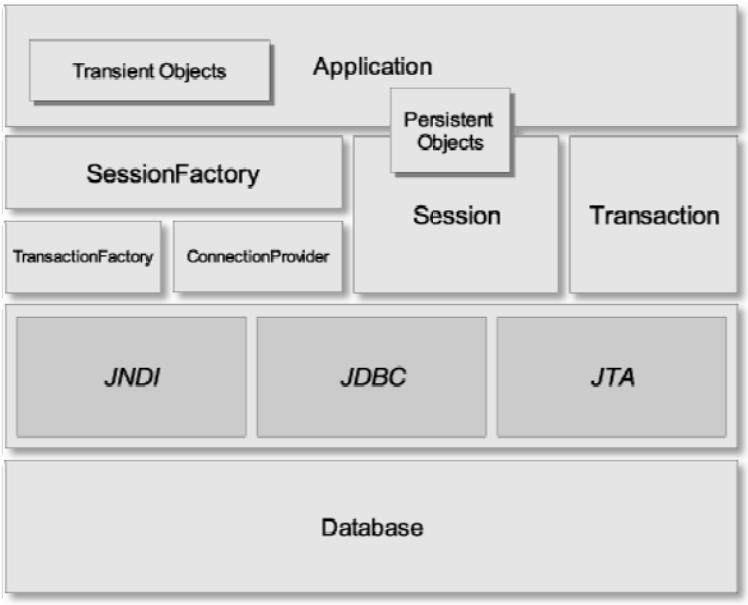

Figure 5.6: Architecture of Hibernate [16]

It consists of the following components:

- **`SessionFactory`** is a thread-safe cache of compiled mappings for a single database. It serves as a factory for `Session` and is client of `ConnectionProvider`.
- **`Session`** is a single-threaded, short-lived object representing a conversation between the application and the DBMS.
- **`Persistent Objects`** are short-lived, single-threaded objects containing persistent state and business function. They are defined by persistent classes and associated with exactly one `Session`.
- **`Transient Objects`** are instances of persistent classes that are not currently associated with a `Session`.
- **`Transaction`** is a single-threaded, short-lived object used by the application to specify atomic units of work, abstracting the application from underlying JDBC or other transactions. A `Session` might span several `Transaction` objects in some cases.
- **`ConnectionProvider`** is a factory for JDBC connections, abstracting the application from underlying database drivers.
- **`TransactionFactory`** is a factory for `Transaction` instances.

Persistent Classes

An instance of a persistent class may be in one of three different states, depending on its association with a Hibernate `Session` object. The states are:

- **Transient**, when the instance is not, and has never been associated with any `Session`. It has no persistent representation in the database.

- **Persistent**, when the instance is currently associated with a `Session`. It has a persistent representation in the database. For a particular `Session`, HIBERNATE guarantees that the persistent identity is equivalent to the JAVA identity (in-memory location of the object).
- **Detached**, when the instance was once associated with a `Session`, but the `Session` was closed. It has a persistent identity and a corresponding representation in the database. For detached instances, HIBERNATE makes no guarantees about the relationship between the persistent identity and the JAVA identity.

When a new instance of a persistent class has been instantiated using the `new` operator, the created object is transient until it is made persistent, using a `Session`. HIBERNATE will then execute the necessary SQL statements automatically.

Instances of persistent classes that have just been saved or loaded using a `Session` are persistent. HIBERNATE will detect any changes made to an object in persistent state and synchronize the state with the database when the unit of work completes.

A persistent object becomes detached when its `Session` is closed. A detached instance can be reattached to a new `Session` at a later point in time, making it (and all the modifications) persistent again.

Database Components

The two implemented components of the thesis for accessing a DBMS are the classes `HibernateDBUtil` and `HibernateDB`. They have already been sufficiently described in Section 4.4.1 but it is necessary to mention them again in order to explain their association with HIBERNATE.

`HibernateDBUtil` is a utility class serving as an interface to HIBERNATE for other components. The convenience class methods it provides mainly offer the possibility to query the database for certain elements of persistent classes. Another task is to create and manage Hibernate `Session` objects. Other classes can call the `currentSession()` and `closeSession()` methods in order to get a reference of or to close a `Session` respectively. The `Session` can be used to communicate with the underlying database.

The responsibility of `HibernateDB` is to convert the data structure of VizzAnalyzer graphs to a set of persistent objects or vice versa in order to save or load graphs to or from database using a Hibernate `Session`. This is necessary, as VizzAnalyzer graphs cannot be mapped directly to database entities without modifying their implementation classes. This, however, lies outside the scope of the thesis.

The next section takes a closer look at the data structure utilized to make graphs persistent.

5.2.2 DBGraph Data Structure

The term DBGraph data structure comprises the persistent class `DBGraph` itself as well as a set of associated persistent classes. Any VIZZANALYZER graph that should be written to database has to be converted to this data structure first. The classes and their attributes are mapped to database tables and SQL data types respectively. Once a graph has been converted to the DBGraph data structure, it can be made persistent by associating the `DBGraph` object with a Hibernate `Session`.

This section will first introduce the classes of the data structure before covering certain issues with it.

Persistent Classes

Figure 5.7 shows a class diagram of all persistent classes the DBGraph data structure comprises.

The diagram shows only a subset of attributes of the implemented classes. It also lacks the getter and setter methods that exist for each attribute. All persistent classes have a unique `id` attribute reflecting the primary key of the corresponding database table. HIBERNATE generates the key automatically when an object is made persistent for the first time.

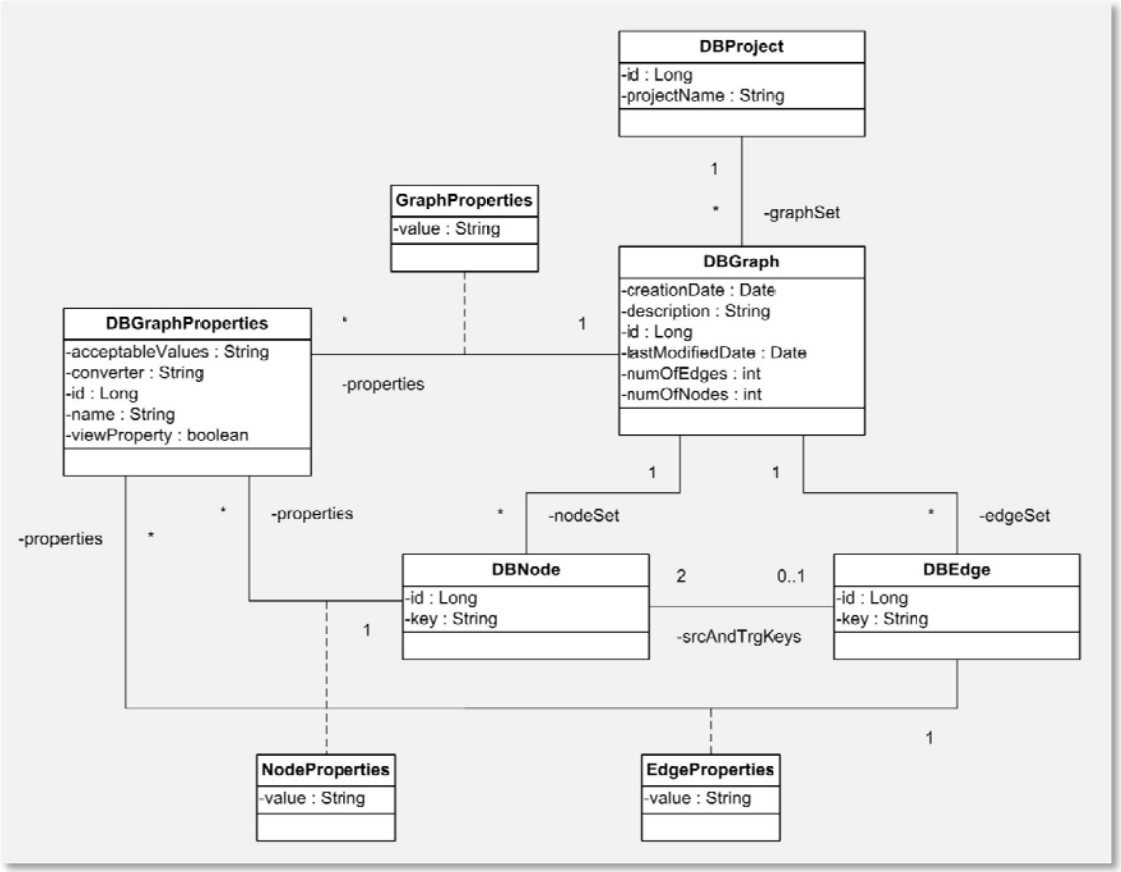

Figure 5.7: Data structure of presistent classes

The three association classes `GraphProperties`, `NodeProperties`, and `EdgeProperties` reflect mappings of `DBGraphProperties` objects to `String` objects; i.e., they contain the values of properties associated with graph elements. The mappings make sense as one instance of `DBGraphProperties`, reflecting a certain property type, can be associated with many instances of graph elements (i.e., graphs, nodes, and edges). The association classes do not exist as real classes as they are realized with JAVA data types that implement the `Map` interface. However, to realize this concept with a relational database a separate table for each kind of mapping is necessary, represented in the diagram by the three association classes.

Table 5.1 shows the relationship between classes of the VIZZANALYZER graph data structure, elements of the DBGraph data structure and database tables.

VizzAnalyzer Graph Data Structure	DBGraph Data Structure	Database Tables
`GraphInterface` class	`DBGraph` class	`DB_GRAPHS`
`NodeInterface` class	`DBNode` class	`DB_NODES`
`EdgeInterface` class	`DBEdge` class	`DB_EDGES`

GraphProperties class	DBGraphProperties class	DB_GRAPH_PROPERTIES
(n/a)	GraphProperties mapping	GRAPH_PROPERTIES
(n/a)	NodeProperties mapping	NODE_PROPERTIES
(n/a)	EdgeProperties mapping	EDGE_PROPERTIES
(n/a)	DBProject class	DB_PROJECTS

Table 5.1: Mapping of classes to database tables

The role of each persistent class of the DBGraph data structure is described in the following:

- The `DBGraph` class reflects graphs consisting of nodes and edges. Each graph belongs to exactly one project and can have an arbitrary number of properties. The attributes shown in the diagram do not store the state of a VIZZANALYZER graph, but rather help to identify graphs stored in a database.
- The `DBNode` class reflects the nodes of a graph. Each node belongs to exactly one graph. Nodes have a unique key and an arbitrary number of properties.
- The `DBEdge` class reflects the edges of a graph. Each edge belongs to exactly one graph. Edges have exactly one source node and one target node, which can be of the same instance. They also have a unique key and an arbitrary number of properties.
- The `DBGraphProperties` class represents types of properties. A graph element can have an arbitrary number of properties but each property must be of a unique type. Each property of a graph element has a String value that may both be empty or `null`. One property type can be associated with many graph elements at the same time. The functionality of this class is described below.
- The `DBProject` class reflects projects stored in a database. Each project has a `String` attribute representing its name. Projects have an arbitrary number of graphs. The class is only used to organize graphs stored inside a database and has no counterpart within the graph data structure of VIZZANALYZER.

GraphProperties Class

The `DBGraphProperties` class reflects the `GraphProperties` class of VIZZANALYZER's graph data structure, used to define types for properties of graph elements. Its functionality and certain issues with it are described in this subsection.

Each element of a VIZZANALYZER graph (i.e., a graph, node, or edge) has an arbitrary number of properties attached to it. The value of a property may be an object of any JAVA class. Code 5.6 shows the declaration of two methods which all graph elements have to implement.

```
public Object getProperty(GraphProperties key);

public void setProperty(GraphProperties key, Object value);
```

Code 5.6: Property methods of graph elements

The `getProperty()` method returns the value of a specific property type or `null` if the property does not exist. Its counterpart, the `setProperty()` method, attaches a value for a specific property type to the element or replaces its value if this property type has already been attached. With both methods, the value of the property is of type `Object`.

On the one hand, this fact makes the graph data structure extremely flexible, as virtually all types of values can be attached to the graph. On the other hand, it also exhibits the problem of how a value of unknown data type can be serialized, i.e., saved to a storage medium such as a file or a database.

The graph data structure of VIZZANALYZER uses a simple type system to addresses this issue, realized by the `GraphProperties` class. It is illustrated in Figure 5.8. Each instance of the class represents a specific property type, stored in its `acceptableValues` attribute of type `Class`. Hence, the type of a value can easily be determined, as `setProperty()` associates each property value with an instance of `GraphProperties`.

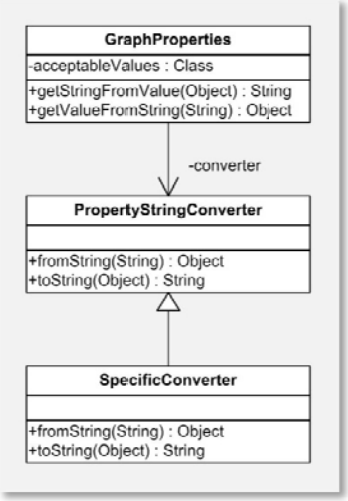

Figure 5.8: Property type system

In order to serialize a property value, the class `GraphProperties` is associated with a converter capable of creating a `String` representation of the value. All converters are actually subclasses of the `PropertyStringConverter` class, offering the methods `fromString()` and `toString()` which are responsible for converting `Objects` to `Strings` and vice versa. The default implementation of the first method simply returns its input `String`. The second method returns with a call of the value's `toString()` method. The subclasses, represented in the diagram as `SpecificConverter`, override these methods in order to implement their own algorithms. `GraphProperties` uses its converter to implement the methods `getStringFromValue()` and `getValueFromString()`.

The combination of these two methods and the `acceptableValues` attribute solves the serialization problem as `String` values can always be serialized. The type of a restored object can be determined by `acceptableValues`.

However, one issue remains, namely how to make the `GraphProperties` class itself persistent. The challenge is that the converter object has to be made persistent as well, but this procedure is not practicable as it is impossible to anticipate new subclasses of `PropertyStringConverter`.

The persistent class `DBGraphProperties` addresses this problem by saving the class name of its associated converter as a `String`. When transforming a `DBGraphProperties` object back to its original `GraphProperties` counterpart, `HibernateDB` restores the converter using the class name and reflection techniques. Thus, the `GraphProperties` object is completely restored and can again be used to restore the property values of the graph elements.

Finally, it should be noted that there is one circumstance under which the restore process of a `GraphProperties` object fails. This situation occurs when the object uses a converter which does not offer a default constructor, i.e., if the constructor requires arguments. In that case, it is impossible to create the converter as the required arguments cannot be provided.

This problem is mentioned here, because it describes a serious design weakness of VIZZANALYZER's property type system. If the implementation of a converter exploits this weakness, the values of the corresponding property type cannot be restored. However, this issue could easily be resolved by refactoring the instance methods `fromString()` and `toString()` of `PropertyStringConverter` to factory methods (i.e., class methods). Thus, recreating the converter objects of `GraphProperties` would be unnecessary.

Further Mapping Issues

There are two additional mapping issues with the DBGraph data structure, which both have the same cause: Similar to the property values of graph elements, the key values of nodes and edges do not have a specified type. But unlike the property values, no type system has been implemented to help serializing the key values.

As a consequence, the key values have to be replaced with unique `String` values during the transformation process, which cannot be restored to their original object types. This represents an acceptable solution, as the keys are not expected to hold any relevant information.

However, the problem shows yet another design weakness of VIZZANALYZER's graph data structure, limiting its practical usefulness. The issue could be addressed by implementing the same type system used for property values or simply by assigning a fixed type to the key values.

5.3 Kenyon Plug-In

The Kenyon plug-in has been implemented to address the main problem of the thesis: the realization of the Retrieval and Analysis Process. The KENYON tool itself plays an important role. It starts the process and controls its procedure. KENYON also handles the communication with the SCM repository and is responsible for the source code retrieval. On the other hand, the `RecoderExtractor` is in charge of controlling the analyses, graph annotation, and storage procedures.

This section inspects both components at greater detail and describes issues with their usage and implementation respectively.

5.3.1 Kenyon

The KENYON tool and its role have already been introduced and described in Section 2.3 and 4.3 respectively. As KENYON is an important part of the solution, this section will go into more detail on the basis of a sequence diagram shown in Figure 5.9. It should be noted that the diagram represents a simplified view on the participants and the messages sent between them. Only relevant elements are included.

The participants in the diagram are:

- the `SCM repository` that will be accessed,
- the `SCM` class serving as an interface to the repository,
- the `Kenyon` class providing a method to start the process,
- the `DataManager` object controlling the source code retrieval, and
- the `RecoderExtractor` object that has been implemented.

The first message in the diagram is a call of the `configureAndRun()` method of the `Kenyon` class, starting the process. An argument specifying the location of the configuration file on the file system has to be provided.

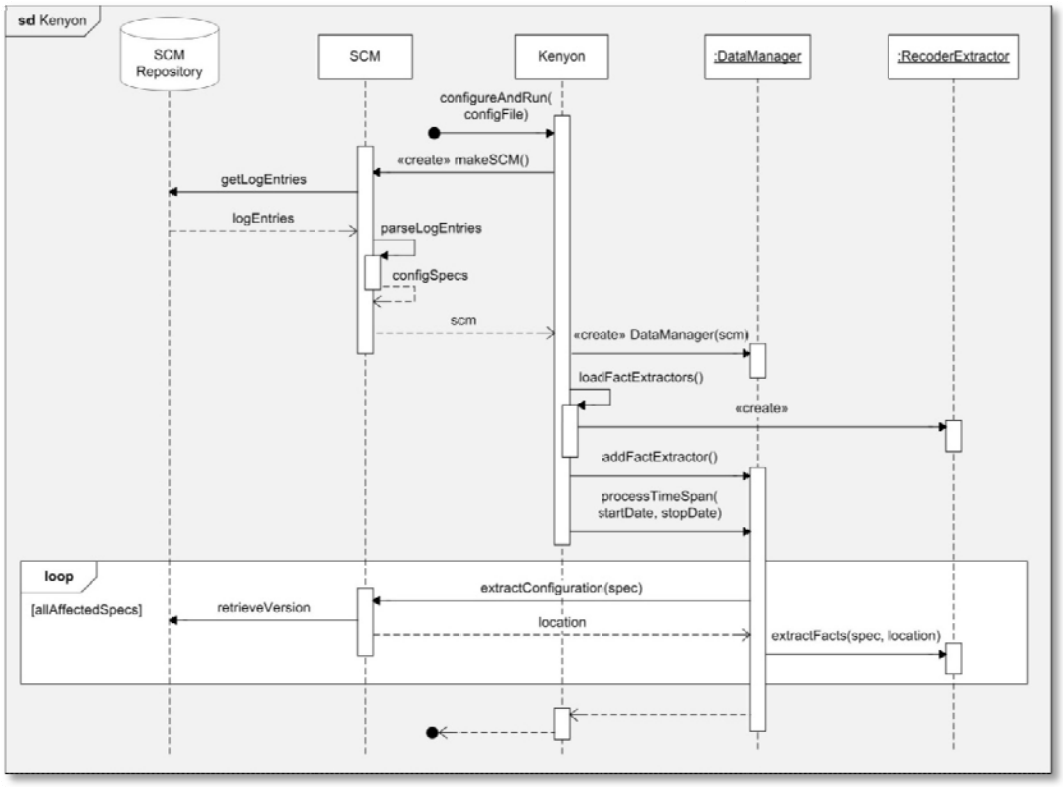

Figure 5.9: Sequence diagram of the Kenyon tool

The next message creates an `SCM` object of the required type by calling the appropriate `makeSCM()` factory method. Code 5.7 shows the implementing code.

```
if (scmType.equals(RepositoryConfig.TYPE_SVN)) {
    scm = SVN.makeSVN(runprops, rc);
} else if (scmType.equals(RepositoryConfig.TYPE_CVS)) {
    scm = CVS.makeCVS(runprops, rc);
} else if (scmType.equals(RepositoryConfig.TYPE_CC)) {
    scm = CLC.makeCLC(runprops, rc);
} else if (scmType.equals(RepositoryConfig.TYPE_FS)) {
    scm = FS.makeFS(runprops, rc);
} else {
    System.out.println("No support for " + scmType + " is currently available.");
}
```

Code 5.7: Creation of a SCM object in Kenyon

It should be noted that all messages to and from the `SCM` class in the diagram are from here on actually sent to and from the created object. During the creation of the `SCM` object, the SCM repository is queried for all its containing log entries. The log entries reflect the complete history of all commit actions that have been performed to check-in files into the repository. The entries include metadata, such as commit dates, file names or revision numbers. This metadata represents the evolutional information which is later used by the `RecoderExtractor` to annotate graphs. KENYON itself uses the log entries to determine which files have to be retrieved from the repository according to the user's settings specified in the configuration file. However, the log entries are specific to each SCM system and therefore they are parsed and transformed into so called configuration specifications, each one representing a single, logical change consisting of one or more commit actions. In the following, the term configuration specification might be referred to as ConfigSpec.

The `SCM` object returned by `makeSCM()` serves as an input parameter for the constructor of the `DataManager` object created in the next step. After that, the Fact

Extractors are created, using the class names provided in the configuration files and reflection techniques. They are subsequently passed to the `DataManager` by calling its `addFactExtractor()` method.

The actual retrieval process starts when the `processTimeSpan()` method of the `DataManager` class is called, having the start and end dates specified by the user as arguments. The method iterates through all ConfigSpecs between the two dates at the user-defined time interval. In each cycle of the iteration, the `extractConfiguration()` method of the `SCM` object is called in order to retrieve the files of the selected ConfigSpec from the repository, storing them in a temporary directory on the local file system. At the end of each cycle, the `extractFacts()` method of each Fact Extractor is called to process the retrieved files. The method receives the current ConfigSpec object and the location of the files as input parameters.

At the end of the diagram, after all affected ConfigSpecs have been processed, the flow of control is returned to the caller.

5.3.2 RecoderExtractor

The `RecoderExtractor` is an implementation of KENYON's abstract `FactExtractor` class. Its task is to control the process of analyzing the retrieved source code, annotating the resulting graphs and finally writing the graphs to the database and the internal graphs storage of VIZZANALYZER.

RecoderExtractor Class

Figure 5.10 and Figure 5.11 show a class diagram and a sequence diagram respectively of the `RecoderExtractor` class. This section will first describe its elements and its role in the Retrieval and Analysis Process on basis of the diagrams before taking a closer look at certain issues with it.

The `FactExtractor` class provides the attribute `scmRepos` to its subclasses, which they can use to get access to all ConfigSpecs. Subclasses have to implement the abstract method `extractFacts()` called by KENYON after a version has been retrieved from the repository.

The `RecoderExtractor` has the following attributes, exclusively accessed by itself and therefore `private`:

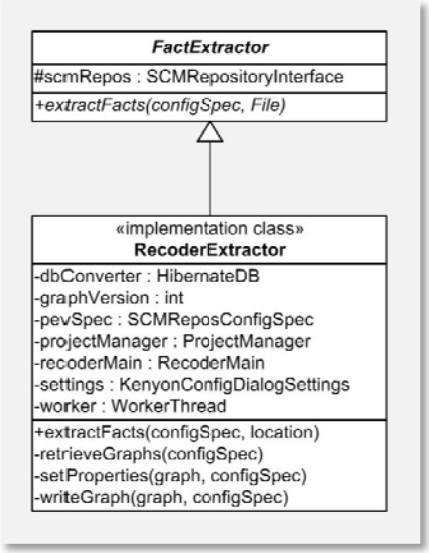

Figure 5.10: The Fact- and RecoderExtractor classes

- `dbConverter` references the `HibernateDB` *object* used for writing graphs to database. The object is a participant in the sequence diagram.
- `graphVersion` is a counter variable used for attaching sequential numbers to all created graphs.
- `prevSpec` references the configuration specification (ConfigSpec) processed in the previous run of `RecoderExtractor`.
- `projectManager` references VIZZANALYZER's `ProjectManager` object used for writing graphs to the internal graph storage. The object is a participant in the sequence diagram.

- `recoderMain` references the `RecoderMain` object of the RECODERCOMP package used for analyzing the retrieved source code and creating graphs. The object is a participant in the sequence diagram.
- `settings` references the Settings object of the Kenyon plug-in's main dialog window containing the user's settings.
- `worker` references the worker thread object reflecting the thread on which the Kenyon plug-in is running.

Some of the attributes are taken on from class attributes of the `Kenyon` wrapper class (cf. Section 4.3.1). Besides the implementation of `FactExtractor`'s abstract `extractFacts()` method `RecoderExtractor` has the following methods, exclusively accessed by itself and therefore `private`:

- `retrieveGraphs()` is responsible for retrieving the graphs created by RECODER and controlling their further processing.
- `setProperties()` is annotating a graph with evolutional information extracted from the ConfigSpecs according to the options set in the Settings object.
- `writeGraph()` is writing a graph to the database and the internal graph storage of VIZZANALYZER according to the options set in the Settings object.

Flow of Events

The sequence diagram is explained in the following subsection. It describes the standard flow of events after `RecoderExtractor` was invoked by KENYON, assuming that no errors occur during its process and no user interaction is required.

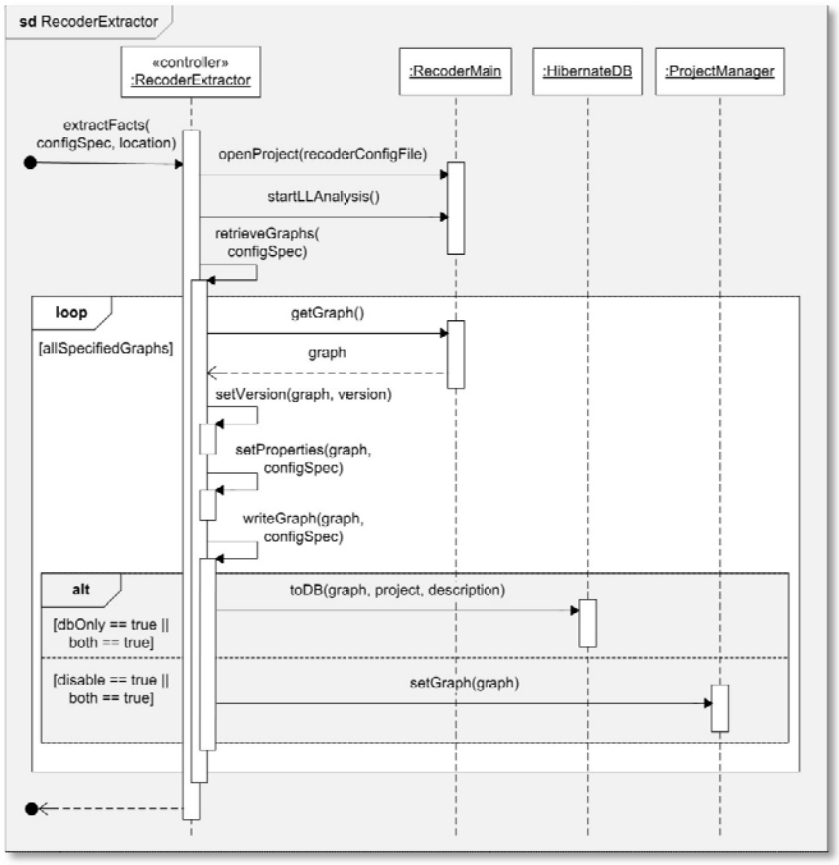

Figure 5.11: Sequence diagram of the RecoderExtractor class

The diagram starts with a call of the `extractFacts()` method controlling the subsequent procedures. First, the RECODER is set up by calling the `openProject()` method with the location to its configuration file, taken from the Settings object, as argument. The analysis and graph generation process, executed by `RecoderMain`, is then started by calling the `startLLAnalysis()` method. `RecoderMain` always creates three standard graphs: The Default LLA Tree Graph, the Class Hierarchy Graph and the Reference Graph.

After the graphs have been created, `retrieveGraphs()` is called to process all graph types that the user specified. The method first fetches the current graph from `RecoderMain` and then attaches the sequential version number to its Label property before annotating the graph with evolutional information, using information of the ConfigSpec object. Finally, `writeGraph()` is called to store the graph to the database utilizing the `HibernateDB` converter, and/or to VIZZANALYZER's internal graph storage by passing its reference to `ProjectManager`.

After all graphs have been processed, the `extractFacts()` method ends and returns to its caller; i.e., KENYON. The diagram also ends here.

Graph Annotation

The graph annotation process, executed by the `setProperties()` method, deserves further elaboration, as there are certain issues with it. The process' responsibility is to annotate the graphs with evolutional information from the SCM repository (cf. Section 4.1). This is done by attaching properties to the nodes of the graph and to the graph itself. The nodes represent classes and other elements, such as interfaces. The graph represents the retrieved version.

Attaching properties to a graph is straightforward, as illustrated by Code 5.8. The example shows how properties are attached to graph elements. In this case, the version's commit date is attached to the graph itself by the `setGraphProperties()` method.

```
private void setGraphProperties(GraphInterface gi, SCMReposConfigSpec currentSpec) {
    if (this.settings.graphPropertiesCheckout == true) {
        gi.setProperty(RecoderExtractor.GRAPH_DATE, currentSpec.getDateString());
    }
    // ... (attach more properties)
}
```

Code 5.8: Attaching properties to graph elements

All graph elements offer `setProperty()`, taking the property type and a value as arguments. Each instance of an element can have no more than one property of the same type. If the property already exists, its value is simply overwritten by the method.

Attaching properties to the nodes of a graph is a more sophisticated process because of two reasons: First, some of the property values must be calculated according to the user's settings; e.g., absolute or relative to the root of the repository. Secondly, the nodes of the graph must be parsed, using their Label properties created by RECODER, in order to attach the right values to them.

The first problem is addressed by keeping a reference to the ConfigSpec of the previously retrieved version, named `prevSpec` (cf. Subsection *RecoderExtractor Class* above). The default settings of the dialog window are, that the values are calculated relative to the previously retrieved version, i.e., from the chronologically first ConfigSpec after `prevSpec`. The values of the first retrieved version are by default calculated absolute to the repository root, i.e., from the chronologically first ConfigSpec stored in the `scmRepos` object. Table 5.2 is a decision table that shows how the reference point

for the calculation is determined in respect to the current version (i.e., first version or not) and the user's settings.

Version / Settings	Criteria	Reference Point / nextSpec =
All Absolute	`if (settings.allAbsolute == true)`	`scmRepos.getFirstConfigSpec();`
First Version / Absolute	`else if (prevSpec == null && settings.FirstVersion == 0)`	`scmRepos.getFirstConfigSpec();`
First Version / Relative	`else if (prevSpec == null && settings.FirstVersion == 1)`	`currentSpec;`
First Version / No Values	`else if (prevSpec == null)`	`currentSpec;`
Relative	`else`	`scmRepos.getNextConfigSpec(prevSpec);`

Table 5.2: Reference point for calculating property values

The reference point reflects the first ConfigSpec object used to calculate the property values. The property values of each relevant node are calculated by iterating through all ConfigSpecs that follow the reference point in chronological order. The properties are updated in every cycle of the iteration by calculating and assigning their new values. The currently referenced ConfigSpec is represented by `nextSpec` and updated at the end of each cycle. Code 5.9 shows how the iteration has been implemented. The method `processSpec()` calculates the new property values and the method `getNextConfig-Spec()` retrieves the chronologically next ConfigSpec object.

```
// Process all ConfigSpecs between the referenced and the current version
while (!nextSpec.equals(currentSpec)) {
     processSpec(nextSpec, nodes);
     nextSpec = scmRepos.getNextConfigSpec(nextSpec);
}
// Process the ConfigSpec of the current version
processSpec(currentSpec, nodes);
```

Code 5.9: Iterating through all ConfigSpecs that have to be processed

So far, it has not been mentioned which properties are attached to the nodes and how their values are calculated. This is explained in Table 5.3.

Property Type	Calculation Method	
	Nonexistent Value	*Preexisting Value*
Author Names	Set current author name	Add current author name if nonexistent in the String
Number of Authors	Set to 1	Increment by 1 if author nonexistent in Author Names
Last Author Name	Set current author name	Replace with current author name
Last Commit Date	Set current commit date	Replace with current commit date

Revision Number	Set current revision number	Replace with current revision number
Number of Commits	Set to 1	Increment by 1
Last Log Message	Set to current log message	Replace with current log message

Table 5.3: Calculation methods for node property values

The second issue with attaching properties to nodes is the parsing process that had to be implemented in order to assign the appropriate values to them. The parsing process matches the Label property of each node with the file names affected by the current ConfigSpec.

The node types relevant for the annotation process are classes, interfaces, and files. The format of their Label property set by RECODER during the creation of the graph is illustrated in Table 5.4. The file nodes reflect the `".java"` files of the analyzed source code containing the definitions of classes and interfaces.

Node Type	Label Format
Class Node	`package.subpackage.ClassName`
Interface Node	`package.subpackage.InterfaceName`
File Node	`package/subpackage/FileName.java`

Table 5.4: Format of node labels

As mentioned in Section 5.3.1, a ConfigSpecs is representing a single, logical change consisting of one or more commit actions. Therefore, they can contain several commit actions of the same author. Each commit action affects at least one file. There are actually differences between various SCM systems, which will not be further discussed here, as KENYON uses these semantics for all supported SCM systems in its `SCMRepoConfigSpec` class defining the ConfigSpec objects. However, the class refers to commit actions as transactions and each affected file is reflected by a revision.

The parsing process is realized by the `processSpec()` method mentioned above (cf. Code 5.9). Its implementation is illustrated in Code 5.10 as pseudo code notation.

```
private void processSpec(configSpec, nodes) {
    transactions = configSpec.getTransactions();
    for (allNodes) {
        nodeLabel = node.getProperty(LABEL);
        // ... (transform label of .java files)
        for (allTransactions) {
            author = transaction.getAuthor();
            log = transaction.getLog();
            revisions = transaction.getRevisions();
            for (allRevisions) {
                revNum = revision.getRevNum();
                date = revision.getDate();
                fileName = revision.getFilename();
                if (file.endsWith(".java") == false) { continue; }
                // ... (transform file name)
                if (nodeLabel.equals(fileName)) {
                    setNodeProperties(node, author, log, revNum, date);
                    break;
                }
} } } }
```

Code 5.10: Parsing process for matching node labels with file names (pseudo code)

The pseudo code shows that both the labels of the node and the file names of the ConfigSpec are transformed before they are compared to each other. The labels are actually only transformed if they reflect files. In this case, the `".java"` extension is cut off and the file-separator character `"/"` is replaced with a `"."` in order to adjust the label to the format of class and interface nodes. The file names are also transformed to this format using a similar approach. However, if the file name does not end with a `".java"` extension, the cycle for the current revision is skipped immediately, as only source code files are of interest.

When the transformed node label matches with the transformed file name, the `setNodeProperties()` method is called with the node object and the extracted information as arguments in order to update the node's properties according to Table 5.3. The method returns after all nodes have been processed.

To conclude the discussion of the parsing process, it should be mentioned that the `processSpec()` method is always processing all of its input nodes rather than finding only those nodes which match to the transformed file names of the ConfigSpec. This is necessary, because the labels of class and interface nodes can be identical with those of corresponding file nodes. Hence, several nodes could be affected by one and the same ConfigSpec revision. In order to speed up the parsing process only relevant nodes are processed, which are by default class nodes, interface nodes and file nodes (cf. Table 5.4).

6 Conclusion and Future Work

This final chapter reflects the results of the thesis. It shows to which extent the problem described in the introduction has been solved and if the goals have been reached. Furthermore, it lists possibilities for future work and research in the context of the thesis. The chapter is concluded by the author's personal view about the topic of the thesis and its outcome.

6.1 Conclusions

The overall goal of the thesis was to provide VIZZANALYZER with the infrastructure for software evolution research. The necessity for software evolution research has been described and motivated in Chapter 1.

The state of the art in this field was subsequently discussed in Chapter 2 by shortly introducing several software tools related to the problem of the thesis. It was concluded that the existing tools do not sufficiently satisfy the goals of the thesis. However, it was suggested that one tool, the KENYON tool, could be integrated into the VIZZANALYZER framework in order to fulfill the goals. Thus, both VIZZANALYZER and KENYON have been described in Section 2.2 and Section 2.3 respectively.

The following two subsections first summarize how the problem of the thesis has been approached before assessing its results.

6.1.1 Summary

The problem addressed by this thesis was:

Customize a front-end plug-in for the VIZZANALYZER framework, providing the capability to extract and analyze multiple versions and evolutional information of software systems from SCM repositories and to store the results.

Section 1.3 concretized this problem by defining the goals of the thesis and criteria used for their validation. In Chapter 3, the goals have been exactly specified by requirements after transforming them into features and use cases.

The realization of the requirements was outlined in Chapter 4. It defined the Retrieval and Analysis Process addressing the problem of the thesis. The overall architecture of the solution was described by introducing the participating components and the dataflow between them. Their integration into VIZZANALYZER as the Kenyon and Database plug-ins was discussed and VIZZANALYZER's plug-in interface was described. The architecture of the common GUI framework, used to realize the GUI of the plug-ins, was explained. The design and architecture was linked to the corresponding requirements and described specifically for each component using class diagrams.

Chapter 5 covered the implementation of the solution by utilizing class and sequence diagrams as well as code examples. The first part of the chapter explained the common GUI framework and the concurrency concept of the GUI in detail. Then, the ORM system HIBERNATE was introduced, responsible for connecting to the database. The persistent data structure, used to store graphs into a database, was described as well as certain mapping issues with the graph data structure of VIZZANALYZER. These issues revealed some design weaknesses within VIZZANALYZER's graph data structure that should be addressed in future work. A solution to these problems was outlined. Finally, the Kenyon plug-in was covered by explaining the functionality of KENYON itself and by describing the implementation of the `RecoderExtractor` class.

6.1.2 Results

The quality of the result can be determined by asserting which of the goals defined in Section 1.3 have been fulfilled. The main goal of the thesis is covered by the Retrieval and Analysis Process (Section 4.1). The process comprises the first three goals of the thesis: Automated source code retrieval of multiple versions of a software system from a CVS or SVN repository (1), automated analysis of the retrieved source code resulting in graphs (2), and annotation of the graphs with evolutional information from SCM repositories (3). Furthermore, the process partly covers the fourth goal: Automated storage of the generated graphs into a database (4).

The Kenyon plug-in (Section 4.3) completely realizes the whole process. KENYON itself (Sections 2.3 and 5.3.1) provides the capability to retrieve multiple versions of software systems from SCM repositories (1). It currently supports the CVS, SVN, and CLEARCASE repositories and thus, it meets the criteria of the goal. The implemented `RecoderExtractor` (Section 5.3.2) for the Kenyon plug-in realizes the rest of the process. It calls the RECODER tool to generate graphs by analyzing the retrieved source code (2) and uses SCM metadata from KENYON for attaching evolutional information to the graphs (3). The graphs are then written to database and/or to the internal graph storage of VIZZANALYZER (4).

The second part of the fourth goal comprises the manual storage of graphs into a database and the ability to manage and restore them. The Database plug-in (Section 4.4) was implemented to meet the criteria. It utilizes the ORM system HIBERNATE (Section 5.2.1) to map the DBGraph data structure (Section 5.2.2) onto database entities. Graphs can be converted to and from this data structure with help of the DB Converter (Section 4.4.1). The converter and the data structure have been integrated as an independent conversion adapter that can be used by other plug-ins as well to read and write graphs from and to database.

Another goal required the implementation of a GUI. This is addressed by the common GUI framework (Sections 4.2.2 and 5.1.1) which both plug-ins use to realize their dialog windows (Sections 4.3.2 and 4.4.2). The framework allows for the usage of concurrency (Section 5.1.2) in order to display progress windows.

A general goal of the thesis was to allow the implemented components to be extendable and reusable. Extendibility is ensured by KENYON and HIBERNATE. KENYON abstracts the access to SCM systems with its `SCMReposInterface`. Support for additional repositories may be provided by future versions or even be implemented by oneself. More functionality can be added via its Fact Extractor and Metric Loader concept. HIBERNATE offers support for almost any relational DBMS. The DB Converter and the common GUI framework have been designed to be reusable by other plug-ins; e.g., the GUI of the preexisting Recoder plug-in was reengineered, using the common GUI framework. The DB Converter is used by both the Kenyon and the Database plug-ins and is available to other plug-ins as well.

Performance was a secondary goal of the thesis. Besides the requirement that frequently executed operations should complete in seconds or minutes, rather than hours or days, no concrete criteria have been defined. Thus, this goal has not been taken under special consideration during the design and development process. Nevertheless, the execution of all ordinary operations completes within seconds for a graph of average size. Operations for managing graphs stored inside a DBMS, especially deleting graphs, can take several minutes for large graphs. These operations could probably be optimized, even though their execution time lies within an acceptable range. Naturally, the execution time of a complete retrieval and analysis process with the Kenyon plug-in greatly varies with the number of retrieved versions and the size of the analyzed system. The tests conducted for the thesis all completed in less than one hour. They also showed that

the implemented solution offers good scalability. Altogether, the criterion of this goal has been met.

This subsection explained in detail to which extent the goals of the thesis have been fulfilled and it proofed that all criteria were met. The summary at the end of Chapter 4 shows exactly by which component each functional requirement was realized and how the non-functional requirements have been addressed. As the requirements themselves were extracted from the goals, it can be concluded that the problem of the thesis has been solved.

6.2 Future Work

The Kenyon plug-in realizes the Retrieval and Analysis Process and thus provides VIZZANALYZER with the basis for software evolution research. Further effort is necessary in order to obtain viable results in this field.

The VIZZANALYZER framework already offers the necessary tools for graph visualization. Hence, the next step could be to implement an analysis plug-in that transforms the resulting data graphs of the Kenyon plug-in into view graphs suitable for visualization, i.e., from base model to model and further on to a view (cf. Section 2.2).

Several graphs, representing different versions of a software system, could be combined to a new graph, showing the evolution of the system. The evolutional information that is attached to the graphs as properties could be used to visualize certain aspects of the system's evolution, e.g., how often each class had been changed between two versions or which developer is responsible for a particular change. Even though the evolutional information from a SCM repository is only detailed enough to detect changes on a file level, a finer-grained level of detail could be achieved. An algorithm could be implemented that compares the Default LLA Tree graphs of two versions, calculating exactly which elements within each class were changed. Such an algorithm would be useful to detect only certain elements that have been affected by changes or to filter out irrelevant elements, such as comments.

Besides future work in the domain of software evolution research, the implemented components themselves could be improved in several respects. The DBGraph data structure serves as an intermediate structure for persistent objects but is basically unnecessary. The graph data structure of VIZZANALYZER itself could be modified in order to support object-relational-mappings. Database connectivity could then be integrated into VIZZANALYZER in such a way that the user can directly see persistent graphs from within VIZZANALYZER's user interface and interact with them.

Another improvement would be to integrate a configuration tab for KENYON into the plug-in's main dialog window, making it unnecessary for the user to manually edit the respective configuration file. This could also include the possibility to query the SCM repository before the actual retrieval process starts in order to get a list of available versions. The user could then directly select the versions that should be retrieved and processed.

Several other suggestions for improvements had been made, e.g., using the Metric Loader concept of KENYON to automatically apply further conversion or analysis steps to the generated graphs (Section 4.2). However, they will not be repeated here as most of them rely on their respective context.

6.3 Personal View

This section gives the author of the thesis the opportunity to express his personal opinion about the thesis.

First, I would like to discuss the title of the thesis. Originally, I planned to implement further high-level analysis plug-ins and visualizations as described in Section 6.2 above. An example of how such plug-ins could work is given in [18], where seven different versions of the VIZZANALYZER have been manually analyzed and merged into one view graph. Hence, the title of the thesis has first been: *"Analysis and Visualization of Software Systems from SCM Repositories"*. The title has been changed when it became clear that the time to implement these features would not be sufficient.

Nevertheless, I am satisfied with the outcome of the thesis and the implemented solution. The Kenyon plug-in allows for easy access to the data stored within SCM repositories and offers users of the VIZZANALYZER the possibility to analyze the evolution of software systems. The biggest advantage of my solution over similar tools is the high flexibility and versatility provided by the VIZZANALYZER framework. Any existing tool which input and output data can be represented as graphs can be integrated into the framework and used together with other tools. This allows software evolution researchers to utilize already existing analysis and visualization tools that would be incompatible otherwise. It is therefore possible to address a wide range of problems within the domain of software evolution research. The currently integrated tools provide a good basis for this purpose. For example, it took me only a few minutes to create a binding that transforms the result graphs of the Kenyon plug-in into view graphs, using the Last Author property to assign different colors to the nodes of the graphs. Figure 6.1 shows a visualization of VIZZANALYZER's Grail package with YED. The color of the nodes indicates that the classes and interfaces have been modified by different authors. First, the visualized version has been retrieved from a SVN repository, analyzed and stored with help of the Kenyon plug-in. Then, I just had to apply the binding to the resulting graph in order to create the view graph displayed in the figure. Since the binding is defined as a XML file, no additional coding was necessary.

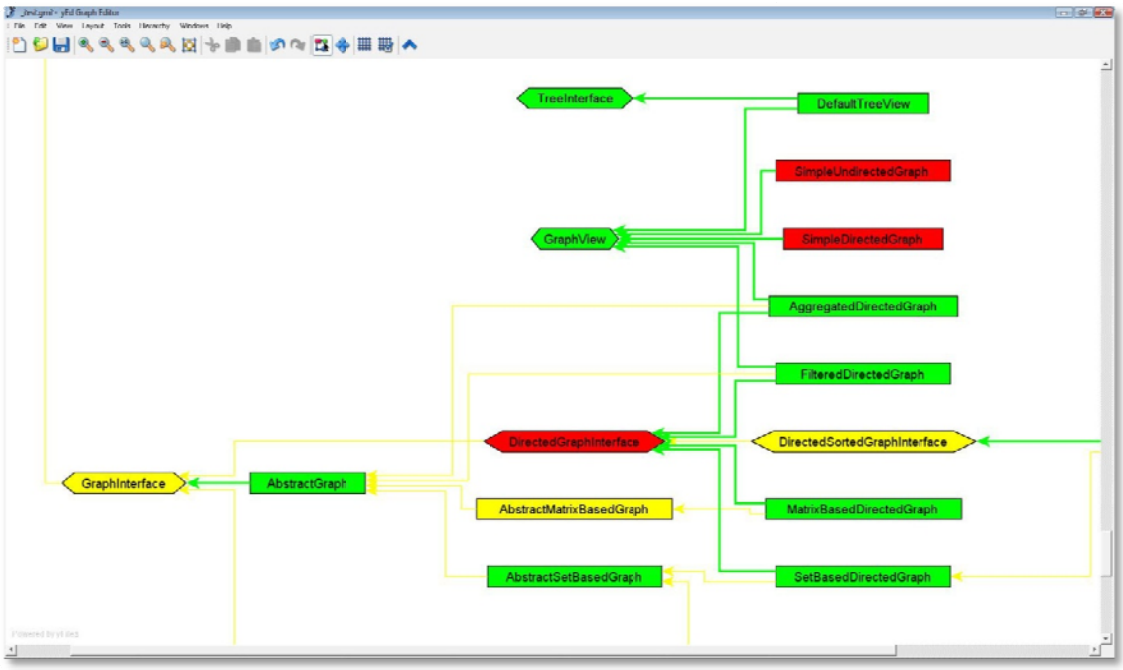

Figure 6.1: Visualization of the Grail package with yEd

I also would like to make some comments about the implementation process. My first step was to create the core functionality of the Kenyon plug-in without database support. This mainly consisted of writing the `RecoderExtractor` class as well as the dialog window classes, which turned out to be a straightforward task. However, matching the

names of files that have been changed in the currently analyzed version with their corresponding node labels during the graph annotation process was not easy to solve (cf. Section 5.3.2). The algorithm that performs the matching had to be tailored to certain characteristics of the RECODER tool and to the currently supported SCM systems. Hence, it might be necessary to adjust the algorithm if further analysis tools or SCM systems are to be supported.

After the Kenyon plug-in was implemented, I continued with the DB Converter component. Using the ORM system HIBERNATE proved to be a good decision, as it simplifies many tasks concerning the connectivity to databases. Nevertheless, it is a complex system that needs to be thoroughly studied first, before it can be used in the right way. Furthermore, creating mappings for persistent classes requires a sound knowledge of relational database systems.

The development of the Database plug-in and its dialog windows then led me to the idea of designing the common GUI framework, based on the dialog window of the Kenyon plug-in, in order to make its architecture reusable for other plug-ins as well.

Finally, I added the `WorkerThread` and progress window classes to keep the GUI responsive to user interaction during long-running operations. The integration of a multi-threading concept turned out to be challenging as well, especially since certain parts of HIBERNATE and the SWING toolkit are not thread-safe. My implementation of the progress window class uses a timer to monitor the status of WorkerThread objects. With hindsight, I would redesign these classes according to the observer architecture, even though the current implementation fulfills the requirements.

My conclusive opinion is that the outcome of the thesis is a useful contribution to the VIZZANALYZER project held by the Department of Computer Science at Växjö University. I hope it can support the evolutional research of software systems.

Bibliography

[1] D. L. Parnas. *Software Aging.* May 1994. Proceedings of the 16th International Conference on Software Engineering (ICSE '94). pp. 279-287.

[2] L. A. Belady and M. M. Lehman. *A Model of Large Program Development. IBM Systems Journal*, Vol. 15(3), 1976, pp. 225-252.

[3] H. Gall, et al. *Software Evolution Observations Based on Product Release History.* Oct. 1997. Proceedings of the International Conference on Software Maintenance. pp. 160-166.

[4] S. G. Eick, et al. *Does Code Decay? Assessing the Evidence from Change Management Data. IEEE Transactions on Software Engineering*, Vol. 27(1), Jan. 2001, pp. 1-12.

[5] VizzAnalyzer. *Applied Research in System Analysis.* [Online] Växjö University, Sweden, Sept. 5, 2005. [Cited: May 25, 2007.] http://www.arisa.se/VA.html.

[6] W. Löwe and T. Panas. *Rapid Construction of Software Comprehension Tools. International Journal of Software Engineering and Knowledge Engineering*, Vol. 15(6), 2005, pp. 995-1025.

[7] *CVS - Concurrent Versions System.* [Online] Derek Robert Price & Ximbiot, 2005-2006. [Cited: March 21, 2007.] http://www.nongnu.org/cvs/.

[8] Subversion. *Tigris.org.* [Online] CollabNet, 2006. [Cited: March 21, 2007.] http://subversion.tigris.org/.

[9] Recoder. *The Recoder Homepage.* [Online] v0.81, University of Karlsruhe, Germany, May 8, 2006. [Cited: March 23, 2007.] http://recoder.sourceforge.net/.

[10] *The School of Mathematics and Systems Engineering.* [Online] Växjö University, Sweden. [Cited: April 30, 2007.] http://www.vxu.se/msi/eng/.

[11] *The IVA Project Homepage.* [Online] GRASE Lab, Dept. of Computer Science, University of California, Santa Cruz. [Cited: May 31, 2007.] http://dforge.cse.ucsc.edu/projects/iva.

[12] J. Bevan and E. J. Whitehead Jr. *Identification of Software Instabilities.* Nov. 2003. Proceedings of the 10th Working Conference on Reverse Engineering (WCRE '03). pp. 134-144.

[13] L. Pekacki and D. Draheim. *Bloof.* [Online] Free University Berlin, Germany, 2003. [Cited: May 31, 2007.] http://bloof.sourceforge.net/.

[14] D. Draheim and L. Pekacki. *Process-Centric Analytical Processing Of Version Control Data.* Sept. 2003. Proceedings of the 6th International Workshop on Principles of Software Evolution (IWPSE '03). pp. 131-136.

[15] O. Alonso, P. T. Devanbu and M. Gertz. *Database Techniques for the Analysis and Exploration of Software Repositories.* May 2004. Proceedings of the 1st International Workshop on Mining Software Repositories (MSE '04). pp. 37-41.

[16] T. Zimmermann. *eROSE.* [Online] Saarland University, Germany, Feb. 17, 2006. [Cited: May 31, 2007.] http://www.st.cs.uni-sb.de/softevo/erose/.

[17] *Eclipse.* [Online] The Eclipse Foundation, 2007. [Cited: May 31, 2007.] http://www.eclipse.org/.

[18] T. Zimmermann, et al. *Mining Version Histories to Guide Software Changes.* May 2004. Proceedings of the 26th International Conference on Software Engineering (ICSE '04). pp. 563-572.

[19] A. T. T. Ying, et al. *Predicting Source Code Changes by Mining Change History. IEEE Transactions on Software Engineering,* Vol. 30(9), Sept. 2004, pp. 574-586.

[20] T. Panas, R. Lincke and W. Löwe. The VizzAnalyzer Handbook. *Applied Research in System Analysis.* [Online] Växjö University, Oct. 2005. [Cited: May 31, 2007.] http://www.arisa.se/.

[21] yEd - Java Graph Editor. [Online] yWorks, 2006-2007. [Cited: May 31, 2007.] http://www.yworks.com/en/products_yed_about.htm.

[22] Vizz3D. *Applied Research in System Analysis.* [Online] Växjö University, Sweden. [Cited: May 25, 2007.] http://www.arisa.se/Vizz3D.html.

[23] T. Panas, J. Lundberg and W. Löwe. *Reuse in Reverse Engineering.* June 2004. Proceedings of the 12th IEEE International Workshop on Program Comprehension (IWPC '04). pp. 52-61.

[24] Graphlet. *The GML File Format.* [Online] University of Passau, Germany, July 20, 1997. [Cited: May 31, 2007.] http://www.infosun.fim.uni-passau.de/Graphlet/GML/.

[25] E. R. Gansner, et al. *A Technique for Drawing Directed Graphs. IEEE Transactions on Software Engineering,* Vol. 19(3), March 1993, pp. 214-230.

[26] Kenyon. *The Kenyon Project Homepage.* [Online] GRASE Lab, Dept. of Computer Science, University of California, Santa Cruz. [Cited: May 31, 2007.] http://dforge.cse.ucsc.edu/projects/kenyon.

[27] J. Bevan, et al. *Facilitating Software Evolution Research with Kenyon.* Sept. 2005. Proceedings of the 10th European Software Engineering Conference (ESEC '05). pp. 177-186.

[28] *Rational ClearCase.* [Online] IBM Corp. [Cited: May 31, 2007.] http://www-306.ibm.com/software/awdtools/clearcase/.

[29] *Hibernate.* [Online] Red Hat Middleware, 2006. [Cited: May 31, 2007.] http://www.hibernate.org/.

[30] *The Java Tutorials.* [Online] Sun Microsystems, 1995-2007. [Cited: May 25, 2007.] http://java.sun.com/docs/books/tutorial/.

[31] *MySQL.* [Online] MySQL AB, 1995-2007. [Cited: July 4, 2007.] http://www.mysql.org/.

[32] T. Panas, R. Lincke and W. Löwe. *Online-Configuration of Software Visualizations with Vizz3D.* May 2005. Proceedings of the 2005 ACM Symposium on Software Visualization (SoftVis '05). pp. 173-182.

[33] WinCvs. *CvsGui.* [Online] 2007. [Cited: July 4, 2007.] http://www.wincvs.org/.

[34] H. C. Gall and M. Lanza. *Software Evolution: Analysis and Visualization.* May 2006. Proceedings of the 28th International Conference on Software Engineering (ICSE '06). pp. 1055-1056.

Appendix User Manual

This chapter contains the user manual of the Kenyon and Database plug-ins describing the system requirements, the installation procedure as well as the user interface of the plug-ins.

A.1 System Requirements

This section describes the system requirements for both hardware and software that are necessary to run the plug-ins. They are based on the system requirements of VIZZANALYZER (cf. *The VizzAnalyzer Handbook* [12]). The development and testing environment is denoted here as well.

A.1.1 Hardware

There is no specific hardware required to run the plug-ins. The recommended hardware to run the VIZZANALYZER is: Pentium 4, 2.6GHz, 1024MB RAM, 3D accelerator 128MB.

The plug-ins have been developed and tested on a Pentium M, 1,5GHz, 512MB RAM.

A.1.2 Software

The Kenyon plug-in requires the installation of a command-line client tool for the respective SCM system that should be accessed. It has been developed and tested with the WINCVS [33] and SVN [8] client tools.

The Database plug-in requires a Database Management System that is compatible with HIBERNATE. A list of all compatible databases can be found at [16]. It has been developed and tested with a MYSQL database [31] of version 4.1.10 (later upgraded to version 5.0.27). Furthermore, a JDBC driver for the used database has to be available. This driver is usually provided by the manufacturer of the database.

VIZZANALYZER as well as the Kenyon and Database plug-ins can run on different operating systems, as they have been developed with Java. However, a Java Runtime Environment (JRE) of version 1.4.2 or higher must be available on the system. The plug-ins have been developed and tested on a Windows XP Professional SP2 system (later upgraded to Windows Vista Business), using the Java 2 Standard Edition (J2SE) version 1.4.2 Software Development Kit (SDK).

A.2 Installation

The Kenyon and Database plug-ins come along with the installation files of the VIZZANALYZER distribution. They are automatically installed together with VIZZANALYZER. The plug-ins are compatible with version 1.0.11a of VizzAnalyzer and above. If an older version of VIZZANALYZER is already in use, it is recommended to upgrade to the latest version.

In order to use the plug-ins, the following steps have to be completed:

1. Run the setup program of the VIZZANALYZER distribution and follow the instructions. Modify the startup scripts to match your environment. More information can be found in the `readme.txt` that comes with the distribution or in [12].

2. Install the client program(s) for the SCM system(s) that should be accessed by the Kenyon plug-in. Make sure that the tool(s) can be run from the command-line.
3. Install and configure the database management system. A catalog that accommodates the tables used by the plug-ins has to be created manually. The database tables will be created automatically by HIBERNATE.
4. Place the JDBC driver file (`.jar` extension) into the `/components/hibernate/` folder under the installation directory of VIZZANALYZER.
5. Configure the file `hibernate.cfg.xml` located in the `/bin/` folder under the installation directory of VIZZANALYZER. More details about this file are given in Section A.4.3 below.

When all these steps have been completed, the plug-ins are ready to be used.

A.3 The Kenyon Plug-In

This section describes how to run the Kenyon plug-in as well as its user interface and the configuration files.

A.3.1 Running the Kenyon Plug-In

To run the Kenyon plug-in VIZZANALYZER must be started and ready for user input. From the menu bar, choose Frontends and select Kenyon (cf. Figure 6.2).

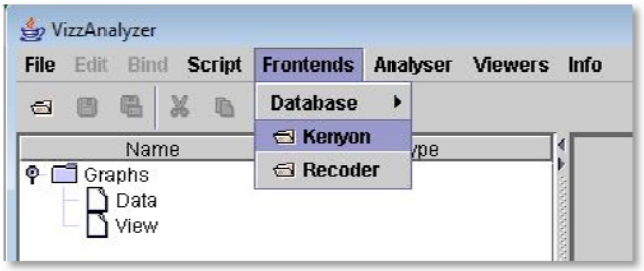

Figure 6.2: Starting the Kenyon plug-in

This opens the main dialog window of the Kenyon plug-in. When the plug-in is invoked for the first time it might take several seconds until the window gets displayed, because the system is trying to initialize the database.

The user has to fill in all required set-up information into the dialog window before the retrieval and analysis process can be started by invoking KENYON. When the process is started the main dialog window closes and is replaced by a progress dialog window. The different dialog windows of the plug-in are explained in the next section.

A.3.2 User Interface

The user interface of the Kenyon plug-in primarily consists of the main dialog window and the progress dialog window. Other windows might be displayed, e.g., when an error occurred or user input is required.

Main Dialog Window

The main dialog window of the Kenyon plug-in allows the user to configure and start the retrieval and analysis process. The dialog window has three different tabs that are displayed in Figure 6.3, Figure 6.4 and Figure 6.5 respectively.

The Cancel button and the Run Kenyon button at the bottom of the dialog window are available from each of the three tabs. The Cancel button closes the window while the Run Kenyon button starts KENYON and the retrieval and analysis process. However, before the process starts the plug-in checks if all required settings have been entered and whether the specified files exist and are readable. If the check fails a message window appears, informing the user about the missing or invalid setting.

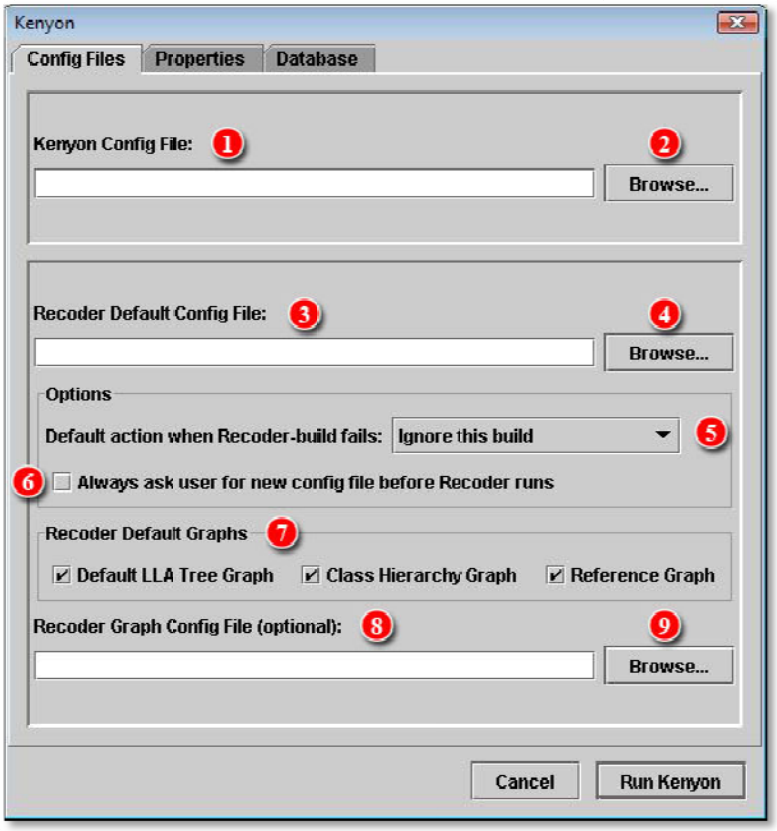

Figure 6.3: GUI elements of the Kenyon dialog window (Config Files tab)

At the Config Files tab, the user has to specify the location of the configuration files for KENYON and RECODER. It is also possible to set some other options for RECODER and the retrieval and analysis process. The elements of this tab are explained in Table 6.1 below.

No.	Description
1	**Kenyon Config File (required)** Enter the location of a configuration file for KENYON or use the Browse button next to the text field to select the file. See Section A.3.3 for a description of the file format.
2	**Kenyon Config File Browse button** Opens a file chooser dialog for selecting KENYON's configuration file.
3	**Recoder Default Config File (required)** Enter the location of a default configuration file for RECODER or use the Browse button next to the text field to select the file. This configuration file will be used by default to analyze each retrieved version. See Section A.3.3 for a description of the file format.

4	**Recoder Default Config File Browse button** Opens a file chooser dialog for selecting RECODER's default configuration file.
5	**Default action when Recoder-build fails** The option selected here will be taken when RECODER fails to analyze a version. The following options are available: • *Ask user for a new config file:* Opens a file chooser dialog. RECODER analyzes the current version again with the selected configuration file. • *Ignore this build (default):* The process ignores the current version and continues with the next version. • *Terminate Kenyon-run:* The whole process is terminated. • *Ask user what to do:* A dialog opens, asking the user to select one of the three options above.
6	**Always ask user for new config file before Recoder runs** If this option is selected, a file chooser dialog is displayed for each version. RECODER analyzes the current version with the selected configuration file.
7	**Recoder Default Graphs** Select the default graph types that RECODER will create during the analysis step. Three types are available: Default LLA Tree Graph, Class Hierarchy Graph, and Reference Graph.
8	**Recoder Graph Config File (optional)** Enter the location of a graph configuration file for RECODER. This allows creating other graph types than the default types. The file contains graph type definitions that RECODER will use for creating the graphs. A description of the file format is not available.
9	**Recoder Graph Config File Browse button** Opens a file chooser dialog for selecting a graph configuration file.

Table 6.1: GUI elements of the Kenyon dialog window (Config Files tab)

At the Properties tab, the user can select the properties that will be attached to the created graphs and their nodes during the annotation process. It is also possible to choose the calculation method of the properties. By default, all properties are selected. However, this step is optional and therefore it is not required to select any property.

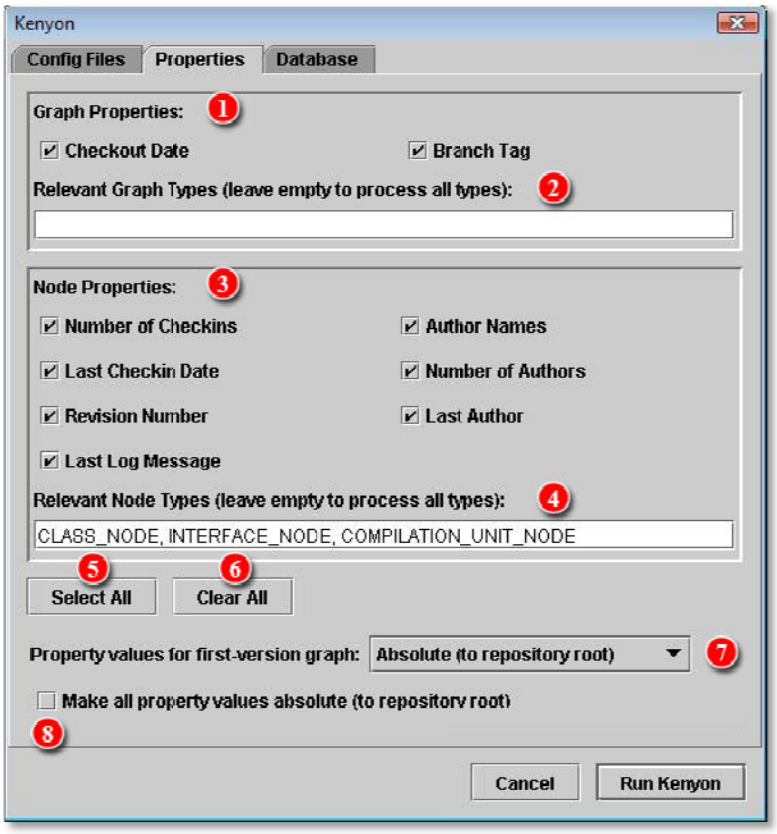

Figure 6.4: GUI elements of the Kenyon dialog window (Properties tab)

The node properties do not apply to all nodes of the graph. It is only possible to attach properties to nodes that can be matched to files of the current version. These nodes usually represent classes, interfaces or files. Furthermore, only those files that have been added, deleted or modified in the current version are considered. The elements of this tab are explained in Table 6.2 below.

No.	Description
1	**Graph Properties** Select the properties that should be attached to the created graphs. The following properties are available: • *Checkout Date:* The date when the version represented by this graph was committed into the SCM repository. • *Branch Tag:* The branch tag under which the version represented by this graph is stored inside the SCM repository.
2	**Relevant Graph Types (optional)** Enter the graph types that should be considered during the annotation process or leave the field empty to process all graphs (the default). If only certain graph types should be annotated with properties, this option can help to increase the processing performance. The entered graph types should be comma-separated and have to match the Type property created by RECODER.
3	**Node Properties** Select the properties that should be attached to the nodes of the created graphs. The following properties are available:

	• *Number of Checkins:* Indicates how often the file reflected by this node was checked into the repository for the current version. The number is calculated either absolute or relative depending on the settings made at No. 7 and 8. • *Last Checkin Date:* The date when the last revision of this file has been checked into the repository for the current version. • *Revision Number:* The revision number of this file for the current version. • *Last Log Message:* The last log message of this file for the current version. • *Author Names:* The names of the authors who modified this file for the current version. They are calculated either absolute or relative depending on the settings made at No. 7 and 8. • *Number of Authors:* The amount of authors who modified this file for the current version. It is calculated either absolute or relative depending on the settings made at No. 7 and 8. • *Last Author:* The name of the author who checked in the last revision of this file for the current version.
4	**Relevant Node Types (optional)** Enter the node types that should be considered during the annotation process or leave the field empty to process all graphs. If only certain node types should be annotated with properties, this option can help to increase the processing performance. The entered node types should be comma-separated and have to match the Type property created by RECODER. By default, this field contains: CLASS_NODE, INTERFACE_NODE, COMPILATION_UNIT_NODE.
5	**Select All** This button selects all properties.
6	**Clear All** This button deselects all properties.
7	**Property values for first version graph** The option selected here determines how the property values of the first version processed during the retrieval and analysis process should be calculated. The following options are available: • *Absolute (to repository root):* The values are calculated from the first revision stored in the repository (this is the default). • *Relative (to previous transaction):* The values are calculated from the version of the previous transaction. • *Don't create properties:* No properties will be attached to the first version.
8	**Make all property values absolute (to repository root)** If this option is selected, all property values will be calculated from the first revision stored in the repository. Otherwise, the values will be calculated relative to the previously processed version (this is the default).

Table 6.2: GUI elements of the Kenyon dialog window (Properties tab)

At the Database tab, the user can set options concerning the storage of created graphs into database. The elements of this tab are only available if the database has been accessible when the dialog was opened. Otherwise, the message “Database not available!” is displayed and the created graphs will be stored inside memory.

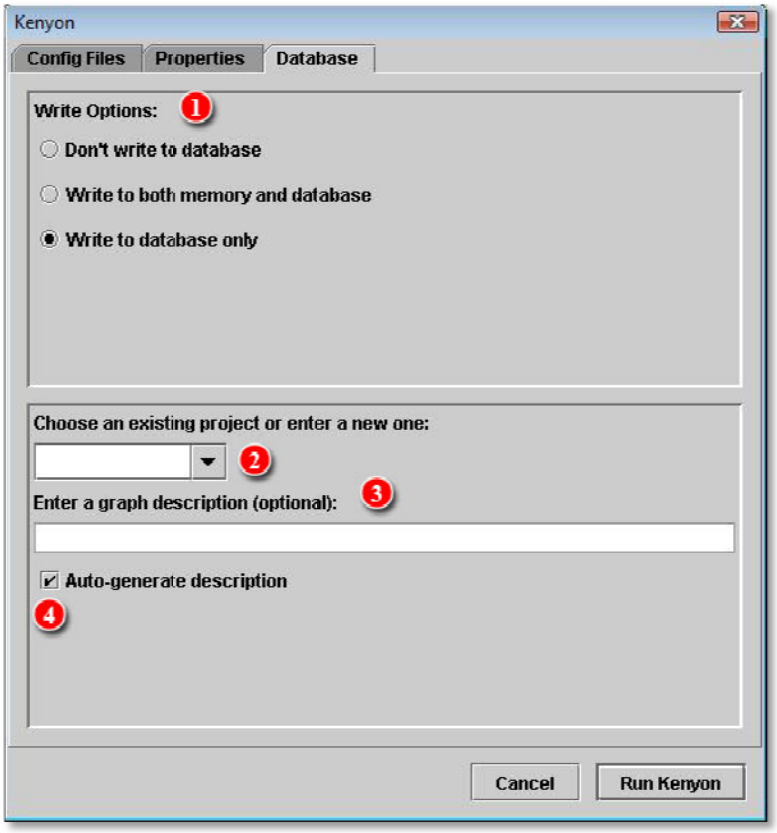

Figure 6.5: GUI elements of the Kenyon dialog window (Database tab)

The elements of the Database tab are explained in Table 6.3 below.

No.	Description
1	**Write Options** The option selected here determines where the graphs will be stored. The following options are available: • *Don’t write to database:* The graphs are stored inside the internal graph storage of VIZZANALYZER but not written to database. • *Write to both memory and database:* The graphs are both stored inside the internal graph storage and written to database. • *Write to database only:* The graphs are written to database but not stored inside the internal graphs storage (this is the default).
2	**Choose an existing project or enter a new one** Select a project from the drop down list or enter the name of a new project into the field. All graphs written to database will be associated with this project.
3	**Enter a graph description (optional)** Enter a description for the graphs that are written to database into the text field. It can help to identify the graphs inside the database but is not obligatory.

4	**Auto-generate description** If this option is selected (the default), graph descriptions are generated automatically. Auto-generated descriptions have the following format: `[last check-in date] :: [project name] :: [graph label]`

Table 6.3: GUI elements of the Kenyon dialog window (Database tab)

Progress Dialog Window

The progress dialog window, shown in Figure 6.6, is displayed after the Run Kenyon button of the main dialog window has been clicked and the retrieval and analysis process was started.

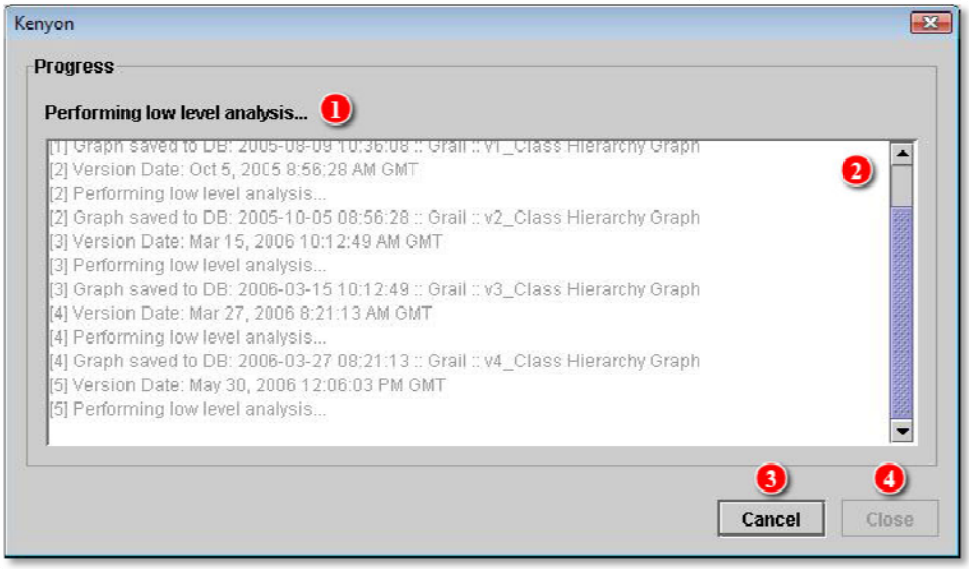

Figure 6.6: GUI elements of the Kenyon progress window

A status message (1) informs the user about the current operation of the process. The complete history is displayed in a text box (2). The numbers at the beginning of each line indicate different versions that have been processed. The process can be stopped by clicking on the Cancel button (3). The window stays visible until the Close button (4) is clicked. This button is only available when the process either has finished or has been stopped by the user.

A.3.3 Configuration Files

The plug-in expects two configuration files, one for Kenyon and one for Recoder, which have to be created and/or modified manually with a text editor.

Kenyon Configuration File

Every invocation of KENYON is configured by a configuration file that sets the parameters for a specific processing run. In this file, the user specifies project parameters, SCM parameters, one or more fact extractors, and zero or more metric loaders. The following sections are taken from the Kenyon User Manual [19] and describe the configuration options in more detail. All filename properties must be absolute.

The configuration files have the extension `.properties`. The default directory of the files is located at `/bin/plugIns/retrievalPlugIns/kenyon`, relative to the root directory of VIZZANALYZER.

Table 6.4 describes the run-specific processing properties. These properties set constraints on the configurations to be processed and the types of data to be stored.

Run-specific Processing Property Configuration	
Required	**Property / Description**
☑	`kenyon.project.id` Identifier of the project being analyzed.
☑	`kenyon.configuration.start.date` Start date in `java.text.DateFormat.MEDIUM` format for the date and the `java.text.DateFormat.FULL` format for the time. The keyword “first” indicates the first configuration.
☑	`kenyon.configuration.end.date` End date in `java.text.DateFormat.MEDIUM` format for the date and the `java.text.DateFormat.FULL` format for the time. The keyword “last” indicates the last configuration.
☑	`kenyon.processing.period` Interval at which configurations should be extracted. Format is [09]+[w,d,h,m,s]. The allowable units are (w)eek, (d)ay, (h)our, (m)inute, and (s)econd. A 10-minute interval would be represented as “10m”.
☑	`kenyon.tmp.dir` The directory in which small, Kenyongenerated temporary files may be placed.
☐ (see note)	`kenyon.usedb` To use only the automated SCM extraction Kenyon subsystem, you will need to specify that the Hibernate database should not be used. Boolean, defaults to “true”. **Note:** For the Kenyon plug-in, this property must be set to “false”.
☐ (see note)	`kenyon.createdeltas` Boolean that indicates if the raw diff output between extracted configurations should be stored as part of the configuration delta. Defaults to "false". **Note:** For the Kenyon plug-in, this property must be set to “false”.
☐ (see note)	`kenyon.storediffs` Boolean that indicates if the raw diff output between extracted configurations should be stored as part of the configuration delta. Defaults to “false”. **Note:** For the Kenyon plug-in, this property must be set to “false”.

Table 6.4: Kenyon run-specific processing configuration properties

Different SCM systems may require different configuration items. Table 6.5 shows the configuration properties for each supported SCM type. Configuring a ClearCase or File System repository requires additional effort which lies outside the scope of this thesis. Please consult the Kenyon User Manual [19] if you plan to use one of these SCM types.

SCM Property Configuration		
Required	**SCM Type**	**Property / Description**
☑	Common	`kenyon.scm.type` "SVN" for Subversion, "CVS" for CVS, "CC" for ClearCase and "FS" for the File System interface.
☐	Common	`kenyon.scm.username` The user name to use for access to the SCM system.
☐	Common	`kenyon.scm.password` The password required for the specified user.
☑	Common	`kenyon.scm.projectname` The name of the SCM project, used for creating unique SCM repository configuration specifications.
☐	Common	`kenyon.scm.branch` Name of the SCM branch that is being processed. Used to create unique SCM repository configuration specifications. Defaults to "trunk".
☑	Common	`kenyon.scm.workspace` The location on the file system in which to place the extracted configuration.
☑	Subversion	`kenyon.scm.url` The URL that identifies the location of the project within the repository.
☑	CVS	`kenyon.scm.protocol` The protocol used for the CVS repository for access, for example "pserver".
☑	CVS	`kenyon.scm.host` The hostname of the machine that hosts the CVS server.
☑	CVS	`kenyon.scm.path` The path to the CVS repository.
☑	CVS	`kenyon.scm.modulename` The name of the CVS module to checkout.
☑	ClearCase	`kenyon.scm.viewpath` The file path to use to access the virtual file system view.
☑	ClearCase	`kenyon.scm.vobtag` A valid VOB tag used to create the snapshot view.
☐	ClearCase	`kenyon.uselabel` Boolean that indicates if labels should be used to create configuration specifications instead of checkin timestamps. Defaults to "false".
☑	File System	`kenyon.scm.file` A configuration file that maps configuration specifications to the directories in which preextracted configurations exist.

Table 6.5: Kenyon SCM configuration properties

MetricLoader configuration properties use symbolic names to represent each specified metric loader. Table 6.6 describes the metric loader configuration items using “[name]” to represent the symbolic name of the metric loader. As the Kenyon plug-in does not use any metric loaders, these properties can be neglected.

MetricLoader Property Configuration	
Required	**Property / Description**
☑	`kenyon.ml.[name].class` Fully qualified class name of the MetricLoader subclass to use.
☐	`kenyon.ml.[name].params.*` This prefix is used to identify the required parameters for the selected metric loader, such as “buildscript.file” and “buildscript.cmd”.
☐	`kenyon.ml.[name].options.*` This prefix is used to identify the optional parameters for the selected metric loader.
☐	`kenyon.ml.[name].constargs` A comma-separated list of properties, defined in this file, that specify a set of arguments for a metric loader constructor. If a constructor with the indicated number of String arguments exists, a MetricLoader will be instantiated with that constructor.

Table 6.6: Kenyon MetricLoader configuration properties

FactExtractor configuration properties use symbolic names to represent each specified fact extractor. Table 6.7 describes the fact extractor configuration items using “[name]” to represent the symbolic name of the fact extractor. The Kenyon plug-in uses only one fact extractor, the class `RecoderExtractor`, which must be configured in every configuration file.

FactExtractor Property Configuration	
Required	**Property / Description**
☑	`kenyon.fe.[name].class` Fully qualified class name of the FactExtractor subclass to use. **Note:** For the Kenyon plug-in, this property must be set to: plugIns.retrievalPlugIns.kenyon.extractors.RecoderExtractor
☐	`kenyon.fe.[name].params.*` This prefix is used to identify the required parameters for the selected fact extractor, such as “buildscript.file” and “buildscript.cmd”.
☐	`kenyon.fe.[name].options.*` This prefix is used to identify the optional parameters for the selected fact extractor.
☐	`kenyon.fe.[name].ml` Comma-separated list of the symbolic names used to identify the metric loaders that are to be assigned to this fact extractor.

Table 6.7: Kenyon FactExtractor configuration properties

Code 6.1 shows an example of a Kenyon configuration file for retrieving source code of an open source project from a public CVS server. Lines that start with a # character are comments.

```
########################################
## Example Kenyon Configuration File ##
########################################

kenyon.project.id freemind

## SCM Configuration
kenyon.scm.type CVS
kenyon.scm.username anonymous
kenyon.scm.projectname freemind
kenyon.scm.branch fm_041017_base_integration
kenyon.scm.workspace C:\\temp\\kenyonws\\freemind
kenyon.scm.protocol pserver
kenyon.scm.host freemind.cvs.sourceforge.net
kenyon.scm.path /cvsroot/freemind
kenyon.scm.modulename freemind/freemind

## Processing Configuration
kenyon.configuration.start.date Jan 2, 2005 00:00:00 AM GMT
kenyon.configuration.end.date Jan 7, 2005 00:00:00 AM GMT
kenyon.processing.period 1d
kenyon.tmp.dir C:\\temp
kenyon.usedb false
kenyon.createdeltas false

## FactExtractor Property Configuration
kenyon.fe.recoder.class plugIns.retrievalPlugIns.kenyon.extractors.RecoderExtractor
```

Code 6.1: Example of a Kenyon configuration file

Recoder Configuration File

The RECODER requires its own configuration file. These files have the extension `.vap` and are using the XML notation. The default directory of the files is located at `/bin/vap_dir`, relative to the root directory of VIZZANALYZER. Table 6.8 describes the elements of a Recoder configuration file.

Element	Subelements / Content	Cardinality
<project>	<projectname> <projectdescription> <projectspecification>	1 1 1
<projectname>	Specify the name of the project.	
<projectdescription>	Specify a description of the project.	
<projectspecification>	<entrypoints> <paths> <filter> <error-tolerance>	1 1 1 0,1
<entrypoints>	<mainclass> <entrypoint> <entryclass> <classload>	0,1 0..* 0..* 0..*
<mainclass>	Specify the class containing the main method.	
<entrypoint>	Methods that can be considered as entry points to the program. The main method specified in <mainclass> is automatically added to the entry points.	
<entryclass>	All public methods in an entry class are considered as entry points.	

<classload>	Classes not directly referenced from the main method that should be loaded anyway.	
<paths>	<sourcepath> <binarypath>	0..* 0..*
<sourcepath>	Directories containing source code.	
<binarypath>	JAR files containing external libraries.	
<filter>	<include> <exclude>	0..* 0..*
<error-tolerance>	Specify a number >= 0 which defines how many errors should be ignored before the analysis is aborted (defaults to 0). The boolean attribute `ignore-unresolved-references` indicates whether unresolved reference errors should be suppressed (not counted as errors). Such errors usually happen if required libraries are missing (defaults to "true").	

Table 6.8: Recoder configuration file specification

Code 6.2 shows an example of a Recoder configuration file. It fits to the example of the Kenyon configuration file shown in Code 6.1 above.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE project SYSTEM "projects.dtd">
<project>
    <projectname>Freemind</projectname>
    <projectdescription>Example Recoder Configuration</projectdescription>
    <projectspecification>
        <entrypoints>
            <entrypoint>freemind.main.FreeMind</entrypoint>
        </entrypoints>
        <paths>
            <sourcepath>C:\temp\kenyonws\freemind</sourcepath>
            <binarypath>$JAVA_HOME\lib\rt.jar</binarypath>
            <binarypath>$VA_HOME\components\junit.jar</binarypath>
        </paths>
        <filter></filter>
        <error-tolerance ignore-unresolved-references="true">5</error-tolerance>
    </projectspecification>
</project>
```

Code 6.2: Example of a Recoder configuration file

Recoder Graph Configuration File

In addition to the specification of a project file, Recoder allows to specify the types of resulting graphs. Various types of graphs are specified by default, e.g., the class hierarchy graph. The user may define other types of graphs by using graph configuration files. These files have the extension `.xml` and are using the XML notation. The default directory of the files is located at `/bin/graph_spec_dir`, relative to the root directory of VIZZANALYZER. Table 6.9 describes the elements of such a configuration file.

Element	Subelements / Content	Cardinality
<graphs>	<graph>	0..*
<graph>	<name> <type> <description> <edges> <nodes>	1 1 1 1 1

<name>	Specify the name of the graph.
<type>	Specify the type of the graph.
<description>	Specify a description of the graph.
<projectspecification>	<entrypoints> 1 <paths> 1 <filter> 1 <error-tolerance> 0,1
<edges>	<edge> 0..*
<nodes>	<node> 0..*
<edge>	Specify the name of the edge.
<node>	Specify the name of the node.

Table 6.9: Recoder graph configuration file specification

Code 6.3 shows an example of a Recoder graph configuration file. It defines a graph of type Interaction Graph with nodes of type ClassDeclaration and InterfaceDeclaration, as wells as edges of the type MethodReference, ConstructorReference, FieldReference, Extends and Implements.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE graphs SYSTEM "graphs.dtd">
<graphs>
    <graph>
        <name>Class Interaction Graph</name>
        <type>Interaction Graph</type>
        <description>Register declared interactions between classes
                     and Interfaces</description>
        <edges>
            <edge>MethodReference</edge>
            <edge>ConstructorReference</edge>
            <edge>FieldReference</edge>
            <edge>Extends</edge>
            <edge>Implements</edge>
        </edges>
        <nodes>
            <node>ClassDeclaration</node>
            <node>InterfaceDeclaration</node>
        </nodes>
    </graph>
</graphs>
```

Code 6.3: Example of a Recoder graph configuration file

More information about the configuration files of RECODER can be found in the VizzAnalyzer Handbook [12].

A.4 The Database Plug-In

This section describes how to run the Database plug-in as well as its user interface and the configuration files.

A.4.1 Running the Database Plug-In

To run the Database plug-in VIZZANALYZER must be started and ready for user input. From the menu bar, choose Frontends and select Database (cf. Figure 6.7). From there, select one of the following tasks: Load Graphs, Manage Graphs, Save Graphs or Save Selected Graph.

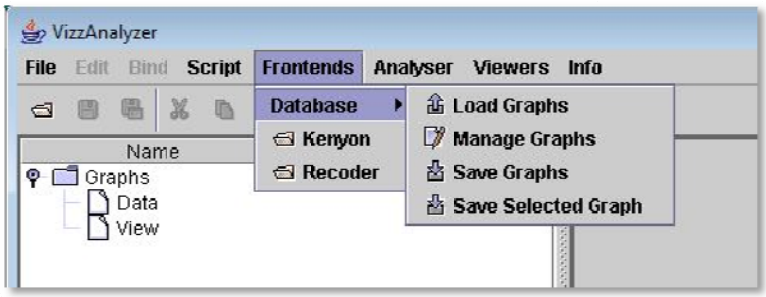

Figure 6.7: Starting the Database plug-in

This opens the corresponding dialog window of the Database plug-in. When the plug-in is invoked for the first time it might take several seconds until the window gets displayed, because the system is trying to initialize the database. The different dialog windows of the plug-in are explained in the next section.

A.4.2 User Interface

The user interface of the Database plug-in primarily consists of three dialog windows for loading, saving and managing graphs. A progress dialog window is displayed for long-running operations. Other windows might be displayed, e.g., when an error occurred or user input is required during an operation.

Load Graphs Dialog Window

At the Load Graphs dialog window, shown in Figure 6.8, the user can select the graphs that should be loaded from database. The dialog window contains three panes: the Projects pane (1), the Graphs pane (2), and the Graph Details pane (3). The Projects pane shows a list of all projects currently stored inside the database. The number of graphs each project contains is shown in parenthesis behind the project name. The user can select one or more projects from the list. The graphs of the selected projects are then displayed inside the Graphs pane, showing the description, the number of nodes and edges as well as the dates of creation and last modification of each graph. The user can select one or more graphs from the list. The Graph Details pane shows a list of all properties attached to the graph that was selected last.

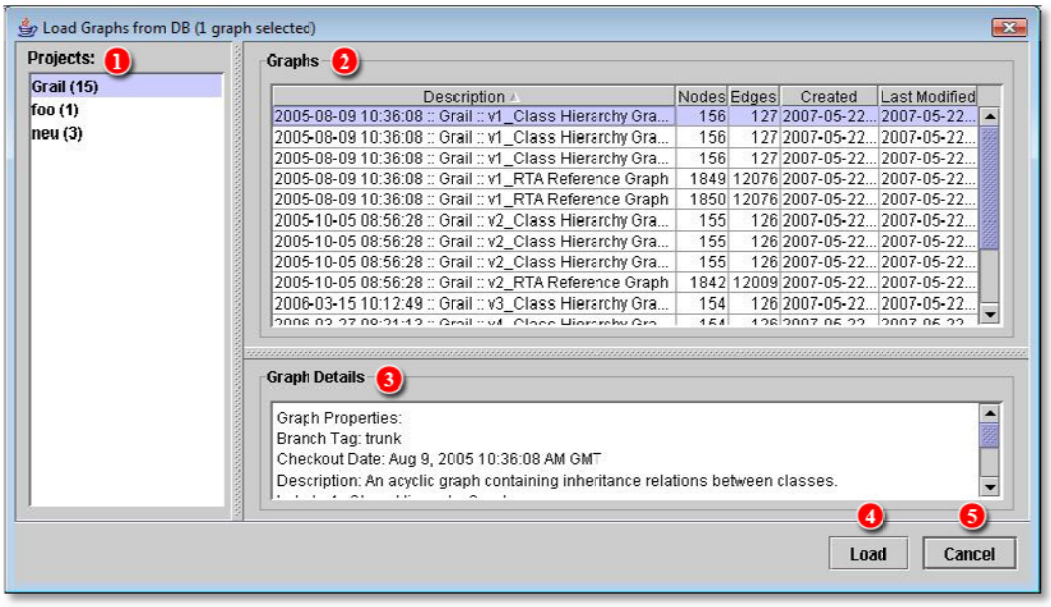

Figure 6.8: GUI elements of the Load Graphs dialog window

When the user clicks on the Load button (4), the Load Graphs dialog window is replaced by a progress window and the selected graphs are being loaded from database. If the user clicks on the Cancel button (5), the dialog window is closed and no graphs are loaded.

Manage Graphs Dialog Window

At the Manage Graphs dialog window, shown in Figure 6.9, the user can manage the graphs stored inside the database. The dialog window contains the same three panes as the Load Graphs dialog window also some additional buttons. At the Projects pane (1), the user can create a new project by clicking on the New button (2), rename a selected project by clicking on the Rename button (3) or delete one or more selected projects by clicking on the Delete button (4). The New and Rename buttons invoke a question dialog, asking for a new project name, while clicking on the Delete button first displays a confirmation dialog before it changes to a progress window.

At the Graphs pane (5), the user can edit the description of a selected graph by clicking the Edit button (6), copy and move selected graphs to another project by clicking the Copy (7) and Move (8) button respectively, and delete selected graphs by clicking the Delete button (9). The Edit button invokes a question dialog, asking for a new description. The Copy and Move buttons both open a dialog from which the user can select the target project before changing to a progress window. Clicking on the Delete button first displays a confirmation dialog before it changes to a progress dialog.

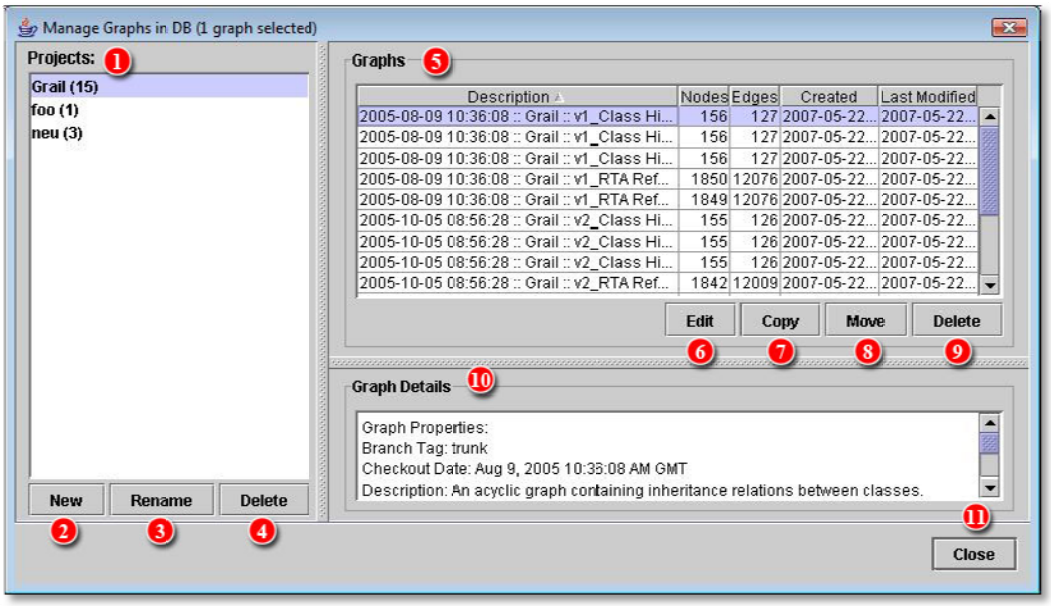

Figure 6.9: GUI elements of the Manage Graphs dialog window

Just like in the Load Graphs dialog, the Graph Details pane (10) shows a list of all properties attached to the graph that was selected last. The Manage Graphs dialog window closes when the user clicks on the Close button (11).

Save Graphs Dialog Window

At the Save Graphs dialog window, shown in Figure 6.10, the user can save graphs currently opened in VIZZANALYZER to database. If the menu entry Save Graphs was chosen, all opened graphs will be written to database, whereas the menu entry Save Selected Graph will only write the selected graph to database. However, for both menu entries the dialog window is the same.

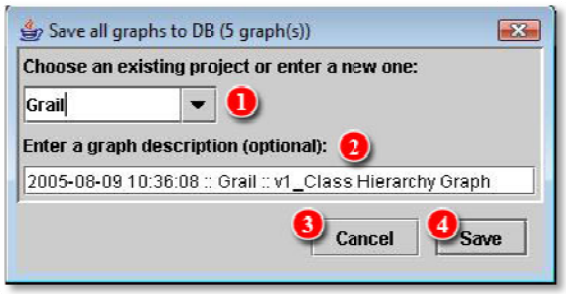

Figure 6.10: GUI elements of the Save Graphs dialog window

The user can either select an existing project from the drop-down list (1) or enter the name of a new project into which the graphs will be written. It is possible to enter a description for the graphs in to the text field (2), but it can be left empty. The dialog window can be closed by clicking the Cancel button (3). If the user clicks on the Save button (4), the dialog window closes and a progress window is displayed instead.

Overwrite Graph Dialog Window

The Overwrite Graph dialog window, shown in Figure 6.11, opens if a graph that is written to database by the Save Graphs dialog already exists inside the database. The dialog shows the description of the corresponding graph (1) and offers several options (2). The user can choose to overwrite the existing graph, either providing a new description for it or keeping the old one. It is also possible to choose to save the graph as a new graph or to not save the graph at all. Additionally, the user can select to apply the chosen option to all following graphs (3). In this case, the dialog window will not open again during the current operation.

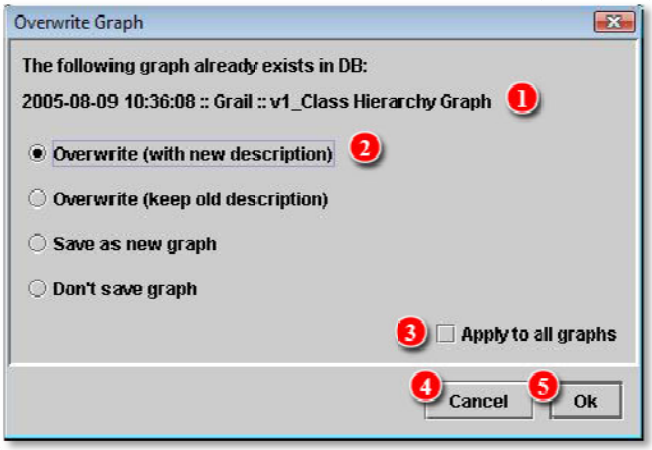

Figure 6.11: GUI elements of the Overwrite Graph dialog window

If the user clicks on the Cancel button (4), the whole operation is stopped and no more graphs are written to database. If the OK button (5) has been clicked, the selected option is applied and the operation continues.

Progress Dialog Window

All progress dialog windows of the Database plug-in have the same layout. Figure 6.12 shows the progress window of the Load Graphs operation as an example.

Figure 6.12: GUI elements of the Database progress window

The progress window shows the current status and progress (1) of the running operation. The user can cancel the operation by clicking on the Cancel button (2). The progress window closes automatically if the operation has finished or if it has been canceled.

A.4.3 Hibernate Configuration File

The Database plug-in requires a configuration file for the HIBERNATE ORM-system. This XML-file must be placed into the `/bin` directory, relative to the root directory of VIZZANALYZER. Code 6.4 shows an example of a Hibernate configuration file.

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hibernate-configuration PUBLIC
        "-//Hibernate/Hibernate Configuration DTD 3.0//EN"
        "http://hibernate.sourceforge.net/hibernate-configuration-3.0.dtd">

<hibernate-configuration>

    <session-factory>

        <!-- Database connection settings -->
        <property name="connection.driver_class">com.mysql.jdbc.Driver</property>
        <property name="connection.url">jdbc:mysql://localhost/kenyon</property>
        <property name="connection.username">test</property>
        <property name="connection.password">test</property>

        <!-- SQL dialect -->
        <property name="dialect">org.hibernate.dialect.MySQLDialect</property>

        <!-- Update the database schema on startup -->
        <property name="hbm2ddl.auto">update</property>

        <!-- mapping files -->
        <mapping resource="grail/converters/hibernateDB/DBProject.hbm.xml"/>
        <mapping resource="grail/converters/hibernateDB/DBGraph.hbm.xml"/>
        <mapping resource="grail/converters/hibernateDB/DBGraphProperties.hbm.xml"/>
        <mapping resource="grail/converters/hibernateDB/DBEdge.hbm.xml"/>
        <mapping resource="grail/converters/hibernateDB/DBNode.hbm.xml"/>

    </session-factory>

</hibernate-configuration>
```

Code 6.4: Example of a Hibernate configuration file

The first four `property` elements contain the necessary configuration for the JDBC connection. The dialect `property` element specifies the particular SQL variant HIBERNATE generates. The `hbm2ddl.auto` option turns on automatic generation of database schemas. The mapping files in the example specify the persistent classes used by the Database plug-in and must be present in all configuration files of the plug-in.

A full description of the Hibernate configuration file is outside the scope of the thesis. Please refer to the HIBERNATE web page [16] for more details on configuring a Hibernate configuration file for your database system.

Matematiska och systemtekniska institutionen
SE-351 95 Växjö

Tel. +46 (0)470 70 80 00, fax +46 (0)470 840 04
http://www.vxu.se/msi/